National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canadä

UNIVERSITY OF ALBERTA

THE IMPACT OF DATA STRUCTURES ON THE PERFORMANCE OF GENETIC-ALGORITHM-BASED LEARNING

BY

LINGYAN SHU ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of DOCTOR of PHILOSOPHY.

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta
FALL 1992

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-77336-7

Canadä

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR:	Lingyan Shu
TITLE OF THESIS:	The Impact of Data Structures on the Performance of Genetic-Algorithm-Based Learning
DEGREE:	Doctor of Philosophy
YEAR THIS DEGREE GRANTED:	1992

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

(Signed) ....

Dated......Sept. 9, 1992......

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **The Impact of Data Structures on the Performance of Genetic-Algorithm-Based Learning** submitted by **Lingyan Shu** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

JSchaeffer
Dr. Jonathan Schaeffer

Joseph Culberson
Dr. Joe Culberson

Kenneth De Jong
Dr. Kenneth De Jong

Renée Elio
Dr. Renee Elio

Curtis Strobeck
Dr. Curtis Strobeck

Dated Sept 11/92

To

my mother Zanru Shen,
the memory of my father Professor Wei-Guang Shu,
and my husband Xuefeng.

ABSTRACT

The amount of knowledge needed, and the way that available knowledge is exploited by a search algorithm, are two important factors that influence the effectiveness and efficiency of a search in a learning system. Blind exhaustive searches require little global information and, in principle, are always effective in finding existing solutions. But the cost of those searches is high. Heuristic searches are efficient but not always effective. Genetic algorithms require little heuristic knowledge about solutions, yet are more efficient than exhaustive searches and are more effective than the heuristic searches when sufficient information is not available.

Premature convergence, sub-solution domination, and internal rule structure establishment and maintenance have been the fundamental problems in genetic-algorithm-based learning (GABL). The existing frameworks either are vulnerable to these problems, or incur a high cost in time and space for their resolution. Recognizing the impact of the data structures in a framework on the resolution of the above problems, this work aims at providing better frameworks in which the internal rule structures are strongly supported by the data structures and operations, and therefore solving or reducing the fundamental problems in GABL. Gaining cooperation and stability through establishing stronger connections is the philosophy presented in this work. Based on this philosophy, two new frameworks for GABL are designed and implemented, in which syntactic and semantic approaches are used respectively to strengthen the internal structures. The analytical comparisons between different frameworks for GABL indicate that the new frameworks encourage co-adaptation of the sub-solutions and stable internal structures with reasonable complexity. The experimental comparisons between the the new frameworks and the most prevalent framework for GABL, classifier systems, have shown that the problems mentioned earlier are effectively reduced and the performance improved significantly.

Genetic learning is a process in which population evolves from one generation to another. In this research, the impact of initial populations on performance is discussed. Two measures for determining how good a population is and two algorithms for selecting better initial populations are proposed. The work on the initial population issue also provides guidelines for incorporating heuristic knowledge into GABL and therefore adding new mechanisms for exploiting available knowledge.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Jonathan Schaeffer for his encouragement and valuable advice, and for his careful reading of every draft of this dissertation, his valuable comments and suggestions. Also thanks to Jonathan Shaeffer for his financial support.

I would like to thank my supervisor committee members Joe Culberson, Renee Elio and Curtis Strobeck for their interest in this work and for their comments. Thanks especially to Joe Culberson for the constructive discussions.

Thanks to the people from the international genetic algorithm community with whom I've had conversations and discussions. To mention just a few of them, I would like to thank Kenneth De Jong, John Holland and Gregory Rawlins for the stimulating conversations. Especially many thanks to Kenneth De Jong for his encouragement, for his valuable comments and suggestions on the draft of this dissertation, and for serving as a member of my dissertation examination committee.

Thanks to the friends who have given me their support. Especially many thanks to Steven Sutphen and Paul Lu for their help in proofreading and producing this dissertation document.

Deep thanks to my grandmother Jie Chen for her example of being an understanding, optimistic and strong person; to my parents for their support and their role model in scientific endeavor; and to my brothers Luchuan and Jianchuan for their encouragement.

Deep gratitude to my husband and best friend Xuefeng Wang. His support and encouragement have greatly contributed to the fruition of this dissertation.

I would like to thank the Faculty of Graduate Studies and Research for the financial support.

Finally, I would like to thank the University of Alberta International Student Center whose services have been very helpful in easing my homesickness when I started my graduate studies in Alberta.

Table of Contents

Chapter 1 Introduction 1

1.1 Genetic Algorithms in Learning Systems 1

1.1.1 Terminology 1

1.1.2 The Role of Genetic Algorithms in Learning Systems 3

1.2 Genetic-Algorithm-Based Learning 4

1.2.1 Frameworks for Genetic-Algorithm-Based Learning 4

1.2.2 Problems with the Michigan and Pittsburgh Approaches 6

1.3 Objectives of this Work 7

1.4 Overview of this Dissertation 8

1.4.1 GA Problems from the Perspective of Data Structures 9

1.4.2 Concepts 10

1.4.3 Hypotheses 11

1.4.4 The Syntactic Approach of Adding Ties 12

1.4.5 The Semantic Approach of Adding Ties 13

1.4.6 Initial Populations 14

1.4.7 Outline of this Dissertation 15

Chapter 2 Understanding the Problems 17

2.1 Genetic Algorithms 17

2.1.1 What is a Genetic Algorithm 18

2.1.2 The Basic Genetic Operators 19
2.1.3 The Basic Execution Cycle of a Genetic Algorithm 21
2.1.4 Performance Measures for GAs 22
2.1.5 Understanding the Advantages and Limitations of GAs 23
2.2 Genetic-Algorithm-Based Learning (GABL) 25
2.3 Genetic Classifier Systems 26
2.3.1 Description of Classifier Systems 27
2.3.1.1 Match-Activation Cycle Walk-Through 30
2.3.2 Properties of Classifier Systems 31
2.4 Problems with Classifier Systems 33
2.4.1 Rule Clustering Problem 33
2.4.2 Rule Association Problem 35
2.4.3 Initial Populations 36
2.4.4 Remarks on the Problem Issue 38
2.5 Comparing GABL with other Machine Learning Paradigms 39
Chapter 3 Impact of Initial Populations on Performance 49
3.1 Examining the Quality of an Initial Population 50
3.2 New Algorithms for Generating Initial Populations 55
3.3 Experiments 56
3.4 Conclusions 70
Chapter 4 Explicit Structural Ties Approach to Classifier Systems 73
4.1 Introduction 73

4.2 Structural Ties Approach 76

4.2.1 HCS Framework 77

4.3 Reducing the Rule Clustering and Rule Association Problems 78

4.4 Experiments with HCS 80

4.4.1 Implementation of the HCS Framework 80

4.4.2 Test Problems and Performance Measures 82

4.4.3 Results and Analyses 85

4.4.4 Discussion 103

4.5 Towards a Unification of the Michigan & Pittsburgh Methods

.... 105

4.6 Remarks on HCS 108

Chapter 5 Implicit Semantic Ties Approach to Classifier Systems 111

5.1 Introduction 111

5.2 VCS Framework 114

5.3 Properties of VCS 119

5.3.1 Uniform Representation 119

5.3.2 Expressiveness 119

5.3.3 Search Space 120

5.3.4 Register Property 123

5.3.5 Build Structures by Building Abstract Relations 126

5.4 Implementation of VCS 127

5.4.1 Test Problems 128

5.4.2 Experiment Results and Analyses 130

5.5 Concluding Remarks 141

Chapter 6 Conclusions 146

6.1 Contributions of the Dissertation 146

6.2 Future Research Directions 148

Reference 151

Appendix 1 157

Appendix 2 159

Appendix 3 161

List of Tables

Table 1. 122

List of Figures

Figure 3.1: Generating a Column Controlled Population. 56

Figure 3.2: Generating a Column and Row Controlled Population. 57

Figure 3.3: Generating an Initial Population with Controls on Columns. 58

Figure 3.4: Generating an Initial Population with Controls on Columns & Rows. 59

Figure 3.5: f_1 performance, population size 48. 61

Figure 3.6: f_1 performance, population size 80. 62

Figure 3.7: f_2 performance, population size 240. 63

Figure 3.8: f_3 on-line performance, population size 64. 64

Figure 3.9: f_3 on-line performance, population size 80. 65

Figure 3.10: f_1 on-line performance by population sizes. 68

Figure 3.11 f_1 on-line, population size 112. 69

Figure 4.1: f_2 on-line performance, ps = 240. 87

Figure 4.2: f_3 on-line performance, ps = 360. 88

Figure 4.3: f_5 on-line performance, ps = 720. 89

Figure 4.4: f_4 on-line performance. 91

Figure 4.5: f_5 on-line performance. 92

Figure 4.6: f_3 on-line performance, ps = 120. 94

Figure 4.7: f_1 on-line performance, ps = 16. 95

Figure 4.8: f_2 on-line performance, ps = 720. 97

Figure 4.9: f_2 instantaneous performance, ps=240. 99

Figure 4.10: f_3 instantaneous performance, ps=360. 100

Figure 4.11: f_3 on-line performance, ps = 120. 104

Figure 5.1: f_1 performance. 132

Figure 5.2: f_3 performance. 133

Figure 5.3: *Nvariable* and *NvariableS* of f_1. 136

Figure 5.4: f_2 performance, $ps = 64$. 137

Figure 5.5: f_2 performance, $ps = 128$. 138

Figure 5.6: f_4 performance, $ps = 64$. 139

Figure 5.7: f_4 performance, $ps = 128$. 140

Figure 5.8: f_1 performance, $ps = 10$. 142

Figure 5.9: f_4 performance, $ps = 16$. 143

Chapter 1

Introduction

This dissertation is concerned with genetic-algorithm-based machine learning systems. In this chapter, first, the role of genetic algorithms in learning systems is discussed. Genetic algorithms are compared with two other representative search algorithms. Then genetic-algorithm-based learning (GABL) is discussed, focusing on the different frameworks for GABL and their problems. Then the objectives of this research are discussed. Finally, an overview of this dissertation is given.

1.1. Genetic Algorithms in Learning Systems

This section shows the significance of genetic-algorithm-based learning. The discussion is focused on the performance differences between genetic algorithms and two other search approaches in terms of efficiency and effectiveness. Informal definitions of important terms or terms with different meanings are given first to clarify their meanings in our discussions.

1.1.1. Terminology

Genetic Algorithms

Genetic algorithms are adaptive search algorithms. They are inspired by genetic evolution processes. Darwin's "survival of the fittest" is the basic principle used. Genetic algorithms search from a population of sample points instead of a single point. The search is directed by the sample population. "Good individuals" in the population have higher probabilities of producing offspring. Individuals that suit the environment stay in the population while the ones that do not fit well into the environment would be replaced by the offspring. Gradually, the population converges towards a population that is dominated by "good individuals". The evolution of the population forms the search process.

Classifier

A classifier is a production-rule-like structure. It contains a condition part and an action part. After the condition is satisfied, the action may be activated. Usually it is represented as a string over an alphabet and has a measure associated with it, called strength or utility. For example a classifier may look like 011ABC/CBB111, where 011ABC is the condition part, CBB111 is the action part and '/' is used to separate the condition and action parts.

Heuristic Knowledge

Heuristic knowledge will refer to additional information about the properties of the problem domain beyond that which is used to describe the problem [Nil83]. Heuristic rules and functions are two forms of heuristic knowledge.

Heuristic Search

Search methods applying heuristic knowledge will be considered heuristic search. In the following discussions, heuristic search does not include genetic algorithms.

Exhaustive Search

Exhaustive search is a search process that considers all possibilities. The exhaustive depth-first and breadth-first searches are examples of this type of search [Nil71].

Local Optima

A local optimum is a point in the search space that is better than all its neighbors but is not the best point in the entire search space.

Plateau

A plateau is an area in the search space, within which all the points have the same value.

1.1.2. The Role of Genetic Algorithms in Learning Systems

There are two important factors that affect the effectiveness and efficiency of a search algorithm in a learning system. The first factor is the amount of knowledge needed by the algorithm. The second one is how the available knowledge is exploited by the algorithm. Exhaustive search algorithms require little knowledge about the solution. In principle, they are always effective in finding existing solutions. However, the lack of knowledge and mechanisms for using knowledge in this kind of algorithms results in poor search efficiency. This inefficiency restricts the effectiveness of this type of search algorithm to problems with a small search space since resources in time and space are usually limited in practice.

Heuristic search has been the most prevalent approach used in artificial intelligence systems since it is more efficient [Nil83]. Heuristic hill-climbing is an example of this kind of search [Nil83, Ric83]. In these search algorithms, heuristic knowledge is used to guide the search and reduce the space that needs to be examined. Two problems may occur with the heuristic searches. One is that the knowledge required by the algorithms is sometimes not available. For example, a hill-climbing algorithm requires knowledge of the direction of the steepest gradient. Soar, which is an architecture for knowledge based systems, is a more concrete example of heuristic search [RLN87]. In SOAR, tasks are formulated as a series of goals and achieving these goals are conducted as searches in the problem spaces. Therefore, knowledge about formulating sub-goals and problem spaces is essential in SOAR.

Another problem with the heuristic search approach is that these algorithms, like other local search algorithms, do not guarantee that the optima, or even just a good solution, will be found. For example, a hill-climbing search is vulnerable to local optima and plateaus. This problem could be dealt with by strategies, such as backtracking, at the price of reduced efficiency. However hill-climbing can be inefficient when dealing with problems with large search spaces and many local optima or plateaus.

As in exhaustive search algorithms, genetic algorithms require little initial heuristic knowledge about the solution. Yet genetic algorithms have better mechanisms for exploiting gained knowledge to guide an efficient search. Compared with heuristic searches, genetic algorithms do not require as much information, but also do not exploit as many different kinds of information. Hence genetic algorithms may not be as efficient as heuristic searches when sufficient knowledge is available. However, in the case of insufficient knowledge or problems with certain properties, such as problems with many local optima, genetic algorithms are more effective and efficient in finding a better solution.

1.2. Genetic-Algorithm-Based Learning

1.2.1. Frameworks for Genetic-Algorithm-Based Learning

The question of how to incorporate genetic algorithms into a learning system is closely connected to the question of how to design or select a framework for the system. Here we use the term of framework rather than representation to emphasize the effects of both the representation and operations. Regarding the representation for a genetic-algorithm-based learning system, two basic issues are considered. First, the sentences in the representation of a framework for GABL should be order independent, i.e. "$Sentence_1$, $Sentence_2$" has the same meaning as "$Sentence_2$, $Sentence_1$". Here, a sentence is defined as a finite length string over an alphabet [HoU69]. Second, the syntax of the representation should be simple. For a representation without the above two properties, operations in genetic algorithms can rarely produce syntactically and semantically correct results [DeJ87].

Two kinds of frameworks that satisfy the above considerations have been used for GABL. One is the production-system-like classifier system, also called the Michigan Approach, in which the basic data structure is a classifier, represented as a string of characters [Gol89a, Hol86, HHN86]. In this approach, each classifier is associated with a strength that tells the utility of the classifier. Single classifiers are the basic operands for the operations in genetic algorithms, meaning that classifiers are used to produce their offspring, namely new classifiers. The other framework for GABL is the so-called Pittsburgh Approach, in which the basic data structure is a program [DeJ88, Smi80]. A program here refers to a set of instructions that can perform the tasks of the system. A program R consisting of a set of production rules is represented as:

$$R = \{R_1, R_2, R_3, ..., R_n\},$$

where R_i $(i = 1, ..., n)$ is a production rule or a classifier, and R can perform the tasks of the system. For example, if '#' means either '0' or '1', {##00/1, ##01/0, ##10/0, ##11/1} is a program that can be used to evaluate the four-variable Boolean function $f(x_0, x_1, x_2, x_3) = ((NOT x_2)\ AND\ (NOT x_3))\ OR\ (x_2\ AND\ x_3)$. '#' is often called the 'don't care' sign. Hereafter, '#' means either '0' or '1'. In the Pittsburgh Approach, the evaluation of "goodness", utility, is associated with the programs instead of the individual classifiers. The utility of a program is changed by evaluating the performance of the program on a learning task. The basic operands for genetic operations are programs, meaning that programs in the population are used to produce new programs.

1.2.2. Problems with the Michigan and Pittsburgh Approaches

Although genetic-algorithm-based learning shows promise, its applications are troubled by a number of problems. Usually GABL systems are used to solve problems where the solution consists of many classifiers. Maintaining good classifiers and the relations that associate these classifiers to each other, while introducing new useful classifiers into the population, has been a problem. This is usually reflected by difficulties in generating and maintaining all the sub-solutions and necessary relationships between the sub-solutions. We term the difficulties rule clustering problem and rule association problem. The rule clustering problem refers to the problem that, as opposed to all the sub-solutions, only a few of them co-exist in the population. This problem occurs because during the competitions, it is possible that a sub-solution with high strength (utility) becomes a super-pattern over the population [Gol89a]. Since classifiers with high strengths are likely to be chosen as objects for genetic operations, gradually the whole population will converge to classifiers with similar patterns. In other words, the result state clusters to one sub-solution. Solving the rule clustering problem is vital to the effectiveness of a genetic learning system. The rule association problem refers to the difficulty in establishing and maintaining relationships among classifiers. The selection algorithms and the parameters used for a task may not be able to establish nor recognize the relationships among classifiers. The behavior of the rule association affects the robustness and stability of genetic-algorithm-based systems. The existing frameworks either are vulnerable to the rule clustering and rule association problems, or incur a high cost in time and space for their resolution.

The Michigan approach emphasizes low-level processing and discovering good rules, which involves extensive exploration. However, in this approach, classifiers are loosely related to each other. Therefore, a genetic algorithm alone is not enough to generate and maintain those relationships among the classifiers and a credit apportionment algorithm has to be used as another mechanism for generating and maintaining those relationships. Nevertheless, the use of such algorithms complicates the systems and introduces new, nontrivial problems. Difficulties in the emergence and maintenance of sub-solution structures in the Michigan type systems have been the incentive for introducing different selection strategies and credit apportionment algorithms, and tuning parameters and operators.

In the Pittsburgh framework, the basic operand for genetic operations is in the form of production programs. In other words, each string in such a system is a concatenation of all the classifiers that constitute a program which can perform tasks. The sub-solutions are connected strongly by the program structure, and a genetic algorithm is sufficient for the emergence of such a structure. Since all the sub-solutions, as single rules, are contained in an individual (a program) of the population, the problem of maintaining these sub-solutions is no longer a difficulty and the issue of assigning credit to single classifiers is avoided in this framework.

Since the basic operand for genetic operations in the Pittsburgh framework is a program, the Pittsburgh approach cannot afford to have a large population of programs due to resource limitations. This restricts the potential for exploration. Hence, the Pittsburgh approach is useful when the emphasis is to find the correct order of a set of rules that constitute a program rather than to find the rules themselves.

1.3. The Objectives of this Work

As discussed in the previous section, the Michigan approach and Pittsburgh approach each have advantages over the other. The Michigan approach (classifier systems) is able to do extensive exploration and therefore pursue low-level learning, but has difficulties in developing and maintaining all the sub-solutions. The Pittsburgh approach has no difficulty in maintaining all the sub-solutions since there is no competition among the rules contained in a program. But the Pittsburgh approach cannot afford to do extensive exploration. Providing genetic learning frameworks that emphasize low-level learning ability (as classifier systems do) yet overcome the problems of classifier systems is the objective of this work. This work tries to avoid introducing new stochastic factors into the system and to reduce the dependence of a system on the credit apportionment algorithms, selection schemes and the related parameters, thus avoiding the difficulties involved with them. Therefore, we look into solutions to the problems in classifier systems from the perspective of data structures.

1.4. Overview of this Dissertation

This work investigates how data structures affect the performance of a genetic learning system. In particular, how different data structures encourage co-adaptation of sub-solutions is studied. We suggest that the robustness of a solution structure in a framework is significantly affected by the data structures and operations. Two frameworks, Variable Classifier System (VCS) and Hierarchical Classifier System (HCS), are designed and implemented to show how to achieve data structures that improve the robustness of a system. In VCS, variables are used to introduce robust data structures. In HCS, imposing hierarchies onto individuals is proposed as a means of gaining better data structures. The analytical comparisons between different frameworks for GABL indicate that the new frameworks encourage co-adaptation of the sub-solutions with reasonable complexity. The experimental comparisons between the conventional classifier systems and the new frameworks have shown that performance improves significantly in these two frameworks.

Genetic learning is a process in which a population evolves from its initial state to a final state when a given performance standard has been met. In this research, the impact of initial populations on a genetic learning system, particularly on the problem of rule clustering is discussed. The experimental results show performance improvement of the initial populations generated by the new algorithms over the ones generated randomly. The work on the initial population issue also provides guidelines for incorporating heuristic knowledge into GABL.

1.4.1. GA Problems from the Perspective of Data Structures

Logical relations and structural relations are two ways of relating classifiers to each other. A structural relation refers to the relationship between classifiers imposed by data structures. To see the reason that classifier systems have difficulties in establishing and maintaining the relationships among sub-solutions, we examine how subsolutions are related to each other in the cases of logical and structural relations respectively.

A solution usually contains classifiers forming a reasoning chain in which the action part of the preceding classifier matches the condition part of the successor. For example, 0000/0001, 0001/0010, 0010/0011 form a reasoning chain, in which the action part of the first classifier "0001" matches the condition part of the second classifier and the action part of the second classifier "0010" matches the condition part of the third classifier. In this case, the classifiers in a chain are related to each other by the logical relationships between them. The generation and maintenance of such a structure in a population are strongly affected by the selection algorithms, credit assignment algorithms and GA operations. Since many parameters are involved in those algorithms and GA operations, the reasoning chain structure is vulnerable to the setting of the parameters.

A default hierarchy is another example of structures built through logical relations. In a default hierarchy, the more general classifiers cover the general situations, whereas the more specific classifiers deal with the exception cases. For example, classifiers ##00 /1 and 1100 /0 form a default hierarchy in which the first classifier covers the situations of 0000, 0100 and 1000, whereas the second one deals with the exception case 1100. It is not difficult to see that the co-existence of the two classifiers in the default hierarchy is important for the system to perform correctly.

Now, we look at an example of structures built through structural relations. A solution of a classifier system usually contains classifiers represented with the '#' character. A classifier, such as ##0111/0, implies classifiers 000111/0, 010111/0, 100111/0 and 110111/0. In this case, the four classifiers whose representation do not contain '#'s are related to each other in the representation with '#'. The '#' representation as a structure links the four single classifiers. One can see that the fate of the four single classifiers is bound together by the '#' representation.

In classifier systems, structures, such as chains and default hierarchies, are essential to the formation of a solution. However, it is difficult to develop and maintain these structures. It is obvious that structures built from structural relations are more robust than the ones built by logical relations. For instance, it is easier to keep all the classifiers related by the '#' character in a population than to keep all the classifiers related by default hierarchy relationship.

1.4.2. Concepts

To study the impact of data structures on the performance of genetic learning, the following concepts are developed in this research.

Primitive Solutions

A primitive solution is defined as a correct response to one and only one environmental or internal state. For example, in classifier systems over the alphabet {'0', '1', '#'}, a primitive solution would be a correct classifier that does not contain any '#'s. A correct classifier containing '#'s would respond to more than one situation and therefore is not considered as a primitive solution.

Internal Structure

An internal structure of a set of elements refers to the relations that impose constraints on to the elements. For example, a vector is a relation that imposes constraints on to its elements. For instance, "0001" and "1001" are two of the elements of vector {0001, 1001, 0000}. In this case, all the constraints imposed on to the elements of a vector apply to "0001" and "1001". Therefore, vector is considered as the internal structure that connects the two elements. Another example is the '#' structure. "#001" is a form of internal structure for "0001" and "1001" since "#001" implies "0001" or "1001". In this case, the two elements are related or connected by the meaning of '#'.

Sub-Solution

A primitive solution or an internal structure of primitive solutions is called a sub-solution.

1.4.3. Hypotheses

Gaining cooperation and stability by establishing stronger connections is the philosophy presented in this work. We suggest that effort to resolve the problems of generating and maintaining internal structures should be given from the perspective of strengthening the connections among primitive solutions through better data structures and operations. This would reduce the dependence of the system on the credit apportionment algorithms, selection schemes and the related parameters, and thus help avoid the difficulties involved with them. Good data structures provide robust internal structures for sub-solutions. Different degrees of robustness of internal structures make the systems require different degrees of support. This is illustrated by the support needed by the Michigan and Pittsburgh approaches. In the Michigan approach, most of the internal structures are based on logical relations. They are not strongly supported by the data structure. Supporting algorithms, such as the credit apportionment algorithm, are essential to the emergence and maintenance of the internal structures. However, in the Pittsburgh approach, internal structures are supported directly by the data structure, namely programs, and the constraint that genetic operations are conducted only on programs. Hence, a credit apportionment algorithm for building structures is not necessary. Our work aims at the evolution of the Michigan approach, i.e. providing better frameworks in which the internal structures of the sub-solutions are strongly supported by the data structures and operations, and therefore providing better genetic learning frameworks which emphasize low-level processing. In this dissertation, the syntactic and semantic approaches are proposed as two ways of adding stronger ties among sub-solutions.

1.4.4. The Syntactic Approach of Adding Ties

In the syntactic approach, the structure of the conventional classifier systems is changed. By introducing hierarchies into the classifier systems, structural ties are imposed onto classifiers.

The basic operand of genetic operations in the conventional classifier systems is a classifier. Each classifier can be viewed as being independent of all other classifiers in the population in that there are no explicit ties between them. This independence can cause a number of problems, such as rule clustering and rule association, which cause system instability, resulting in poor performance. By imposing structural ties between classifiers, we force dependence of classifiers on each other and, as a result, coherence, cooperation, and co-adaptation are introduced to the system. A hierarchically structured classifier system (*HCS*) has been designed and implemented to show the effect of this structuring [ShS91]. In *HCS*, at the lowest level, classifiers (*individuals*) are grouped into *families*. For large and complicated problems, higher-order structures, such as a *community of families*, may be needed.

This structuring has two effects. First, individuals in a family cooperating with each other can yield a strength that is stronger than the sum of that of the individual's. Second, stronger individuals in a family can help the weaker ones. This results in the effect that individuals that have not been tested enough can stay longer in the population. Therefore, co-adaptation is supported and over-domination is discouraged.

1.4.5. The Semantic Approach of Adding Ties

In this semantic approach, ties are imposed by semantic constraints. Learning is a process of establishing relationships. This requires a learning system to be able to represent the emergence of these relationships or internal structures. Good data structures in a framework result in strong and stable internal structures. A variable is a data structure whose semantics defines a relation over its instances. This characteristic of variables makes them essential to symbolic processing. We introduce variables into classifier systems to improve the expressiveness of the systems and strengthen the connections between primitive solutions. The use of variables enhances genetic classifier systems with the ability of symbolic learning systems, therefore narrowing the gap between genetic-algorithm-based learning systems and symbolic learning systems. With variables, built-in knowledge, knowledge models and conceptual hierarchies become possible. This enables us to combine the advantages of the symbolic learning systems and genetic-algorithm-based learning systems. Since many solutions can be expressed in a succinct way, the number of sub-solutions contained in a solution is smaller. This implies that the connections among the primitive solutions are stronger. Therefore, the problems of rule clustering and rule association are reduced effectively. Variables are essential to information processing and problem solving. The lack of variables in the neural network approach is considered a major deficiency [Nor86]. In this work, a new framework, Variable Classifier System (*VCS*), is designed and implemented, in which classifier systems are evolved to include variables [ShS89]. The result is that variables act as implicit connections between the instances being represented by them. For many problems, *VCS* provides stronger internal primitive solution structures, thus allowing those problems to have their solutions expressed in a succinct manner, reducing the amount of work required by genetic search and credit assignment algorithms. A common characteristic of *HCS* and *VCS* is that, at early stages of an evolution, it is hard for a single classifier to be dominant. Therefore, classifiers containing various schemata are kept in the population. In *HCS* and *VCS*, the probability of a schema being deleted from the population without being tested adequately is small.

1.4.6. Initial Populations

Genetic learning is a process of evolving the initial population. In this research, the impact of initial populations on a genetic learning system is discussed. Particularly, we studied how more schemata can be included into an initial population. This is examined from two perspectives. One is the "background information" contained in an initial population, the other is the internal structures contained in an initial population. If the background knowledge is biased to certain primitive solutions, the population tends to converge prematurely to the sub-solutions representing these primitive solutions. As well, different initial structures cost differently to evolve to the solution structure. The performance using an initial population whose structures are "close" to the solution structures would be more efficient.

To guide the selection of an initial population, two measures are defined based on the above considerations. Two theorems are developed using these two measures. Based on these two theorems, two algorithms are presented for generating better initial populations to maximize the potential of a classifier system. Initial populations influence the common genetic algorithm problems of premature convergence and rule clustering. A bad initial population tends to either converge fast at the beginning before the correct solutions are generated, or take a long time to generate the correct sub-solutions. In the literature, initial populations are usually chosen randomly. Our research suggests this may be a dangerous practice. The significance of this study can also be seen from its function in guiding the incorporation of heuristic knowledge into a genetic learning system. Genetic-algorithm-based learning systems are general learning systems where little domain specific heuristic knowledge is required. This property enables genetic learning systems to be effective for problems with insufficient information. Nevertheless, the efficiency can be improved by a good use of heuristic knowledge. However, available heuristic knowledge would be ignored by a genetic learning system because there is no mechanism for incorporating it into the system. Grefenstette discussed this problem and proposed several strategies based on the context of the traveling salesman problem [Gre87]. This work provides more general guidelines on this issue.

1.4.7. Outline of this Dissertation

This dissertation contains six chapters.

Chapter 2 provides background knowledge on genetic algorithms and genetic-algorithm-based learning. Problems in genetic algorithms and genetic learning, and the existing solutions are discussed. As well, comparisons are made among genetic-algorithm-based learning, neural network learning and symbolic learning approaches.

Chapter 3 discusses the importance of an initial population on the performance of a classifier system. Two new algorithms are proposed for generating good initial populations.

In chapter 4, a syntactic approach for gaining better data structures is proposed. In this approach, structures are imposed on to classifiers which are the basic operands of genetic operations in a classifier system. A hierarchically structured classifier system (*HCS*) is designed and implemented to show the effect of this structuring [ShS91]. At the lowest level, classifiers (*individuals*) are grouped into *families*. The relationships between the *HCS* framework and the Michigan and Pittsburgh approaches for classifier systems are discussed.

In chapter 5, a semantic approach for gaining better data structures is proposed. In this approach, variables are introduced into classifier systems. With the use of variables, many solutions can be expressed in a succinct way. This usually reduces the number of sub-solutions needed for a solution. In this work, Variable Classifier System (VCS), is designed and implemented, in which classifier systems are evolved to include variables [ShS89]. VCS shows significant improvements over the conventional classifier systems.

Finally, Chapter 6 makes the concluding remarks on the contribution of this dissertation and related future research directions.

Chapter 2

Understanding the Problems

This chapter first provides background knowledge on genetic algorithms and genetic algorithms based learning. Then classifier systems are described and analyzed in detail, which include the properties of classifier systems, the problems and solutions. Finally, relationships between genetic-algorithm-based learning and two other learning paradigms, symbolic learning and neural network approaches, are discussed.

2.1. Genetic Algorithms (GAs)

Genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics. Since the late 1960's, genetic algorithms have been studied intensively by John Holland and his colleagues [Bet81, DeJ75, Gol89a, Hol75, Hol86]. In his important book, *Adaptation in Natural and Artificial System*, Holland analyzed mathematically the robustness of genetic algorithms and proposed the influential theory of schemata [Hol75]. Other theoretical work includes studies on parameters involved in genetic algorithms, such as population sizes [Gol89b]. Other than the pure mathematical analyses, genetic algorithms have been studied mainly in the contexts of function optimization and adaptive learning systems. Studies have shown the superiority of genetic algorithms as a function optimization tool in dealing with certain types of optimization problems, such as multimodal and high-dimensional functions [Bri81, DeJ75]. As a discovery mechanism, genetic algorithms have been used in many learning systems. It has been shown that genetic learning systems work well with dynamic, noisy and nonlinear problems, and provide a model for low level processing [Gol83, Rio86, ScG85, Smi80, Wil86].

Genetic algorithms and their applications have drawn increasing attention from many areas such as dynamic control, biological evolution, and artificial intelligence [Gol89a].

2.1.1. What is a Genetic Algorithm?

Genetic algorithms are derived from a computational model of evolutionary genetics. Darwin's survival of the fittest is the basic principle used there. Individuals that suit the environment stay in the population while the ones that do not fit the environment would be replaced by new ones. The model is based on the following set of assumptions: chromosomes are strings with a finite length l, a population contains a fixed number of chromosomes and each chromosome has a fitness measure associated with it, which indicates its ability to survive and produce offspring. The population is a dynamic entity. New individuals are continually being generated to replace existing ones. The time elapsed between two successive populations is considered a generation. For simplicity, in our discussion, all the strings in a population have the same length of l.

Schema is an important concept when we discuss the search properties of genetic algorithms. A schema is a compact string representation for a set of strings. Usually, symbol '*', meaning either '0' or '1' in a representation framework over alphabet {'0', '1'}, is used in representing schemata. For instance, "*10*" is a schema which defines four strings: "0100", "0101", "1100" and "1101". The characteristics of a schema are determined by its positions with fixed values, as opposed to its '*' positions. In a representation over alphabet {'0', '1'}, a fixed value is either '0' or '1'. Genetic algorithms search for good strings through searching for schemata. This is what makes a genetic algorithm different from an enumerative or a random search. The following example shows how schemata are combined to build new ones. "1*1**0" and "0**011" are two schemata and represent the set of strings

{ 101000,
101010,
101100,
101110,
111000,
111010,
111100,
111110,
000011,
001011,
010011,
011011 }.

If we combine the first three bits of the first schema with the last three bits of the second one, we get a new schema "1*1011", which is different from, and not included in, the two parent schemata and represents two new strings "101011" and "111011". The schemata work as building blocks for the construction of new strings. This example illustrates the crossover operation of genetic algorithms, which will be discussed next.

2.1.2. The Basic Genetic Operators

Reproduction, *crossover* and *mutation* are the basic operators used in genetic algorithms. They are functionally sufficient in the sense that they are able to generate all individuals of the search space. In the following, strings of length 6 over the alphabet {'1', '0'} are used as examples to describe the genetic operators and their functions.

Reproduction

Reproduction is an operator for copying strings (individuals) according to their fitness from one generation to another. For example, consider the following population:

$$\begin{aligned} S_1 &: 000000 \quad \text{(fitness 0.25)} \\ S_2 &: 010100 \quad \text{(fitness 0.50)} \\ S_3 &: 001100 \quad \text{(fitness 0.25)} \\ S_4 &: 101000 \quad \text{(fitness 0.00).} \end{aligned}$$

A reproduction operation that copies each string with a probability in proportion to its fitness would produce the following population:

$$\begin{aligned} S_1 &: 000000 \\ S_2 &: 010100 \\ S_2 &: 010100 \\ S_3 &: 001100. \end{aligned}$$

There are two copies of S_2 and no S_4 in the new population. It can be seen that such reproductions favor individuals with higher fitness. Therefore, "good" individuals stay in the population generation after generation, and "bad" ones become obsolete and are removed from the population eventually.

Crossover

The crossover operator is the primary operator for generating new strings and instances of new schemata. A simple crossover operation is conducted in three steps: first, pick up a pair of strings from the population as the parents; second, select a position between two characters of the strings and split each string at that position into two segments; third, swap the segments between the two strings and get two new strings as the children. For example, strings "111000" and "011111" are selected from a population and the position between the third bit and fourth bit is chosen. Then the strings are split, producing segments "111" and "000" from the first string and "011" and "111" from the second one. After swapping the segments, two new strings "111111" and "011000" are created. This example shows how crossover produces new individuals. It is also important to see how instances of schemata are added into a population. Considering schemata "*11000" and "011**1". One of the strings used in the above example, "111000", is an instance of "*11000", while the other string is an instance of "011**1". One of the children of the above crossover, "011000", is another instance of "*11000" while the other child, "111111", is not an instance of either of the schemata, but is an instance of the new schema "1****1". This illustrates how existing schemata are reinforced and new schemata are explored.

The crossover operator samples both familiar and unexplored regions of the search space. Other than the single point crossover, other variants, such as two points crossover in which two positions are selected in a string to separate the string into three segments, are used as well.

Mutation

The mutation operator is required since reproduction and crossover can only manipulate what is available in the sample population. If a value at a particular string position is missing from the population, with only reproduction and crossover operations, the search will never reach the strings with that value at that position. For example, if value '0' does not appear in the second position of any string in the population, the search will never create instances of "*0****". Mutation refers to changing the value at a random string position to another value. The result might be a new useful string, or possibly one with little benefit to the system. Mutating a string is a background pressure that guarantees no value will be permanently lost from the "pool" of values available to the genetic algorithm.

2.1.3. The Basic Execution Cycle of a Genetic Algorithm

There are a variety of ways to implement genetic operations. The following is a basic genetic algorithm:

1) Generate an initial population.

2) Evaluate the population. If it satisfies a given standard, stop the execution.

3) Carry out crossover operations with a given probability. Replace some string(s) with low fitness in the population with the new offspring.

4) Carry out mutation operations with a given probability. Replace some string(s) with low fitness in the population with the new offspring.

5) Go to Step 2.

The successive populations of strings designate a search trajectory through the search space. If the reproduction rate for each string is proportional to its fitness (strength, performance, fitness are equivalent here), the strings that remain in the sample population, over time, are those proven to be the most fit. The search trajectory is steered toward the individuals or regions in the search space with above average fitness. The effect of the above procedure is to emphasize various combinations of schemata as building blocks for the construction of new good strings.

2.1.4. Performance Measures for GAs

DeJong in his Ph.D. thesis experimentally examined the behavior of GAs [DeJ75]. Two kinds of performance measures were proposed. The first one is called *on-line performance*, which is defined as

$$P = \frac{\sum_{t=1}^{T} f(S_t)}{T}.$$

The second performance measure is called *off-line performance*, defined as

$$P = \frac{\sum_{t=1}^{T} \max_{i=1}^{t} f(S_i)}{T}.$$

Here T is the current generation, S_t is the string generated in generation t, and $f(S_t)$ is the function evaluation for S_t. By the definitions, it can be seen that the on-line performance emphasizes the whole process, while the off-line performance emphasizes the maximum value achieved. On-line performance is usually the more important measure in learning systems.

2.1.5. Understanding the Advantages and Limitations of GAs

Recombination of building blocks (schemata) is an important characteristic of genetic algorithms. Genetic systems can respond to any situation by recombining a set of relevant strings. The recombination enables genetic algorithm systems to avoid using a distinct string for each situation. This is an important property for induction systems dealing with large search spaces. In those situations, the systems work on samples of the search spaces, and it is impossible or costly to represent every point in the search space beforehand.

It has been shown that in a genetic-algorithm-based classifier system with M rules, as the genetic algorithm generates new rules, it is effectively selecting amongst more than M^3 building blocks [Hol86]. The following discussion shows how genetic algorithms work effectively and efficiently.

To see the advantages of the genetic algorithms, it is vital to understand the so-called ***implicit parallelism***. Assume C represents the space of all possible strings with length l over the alphabet {'0', '1'}. A genetic algorithm always considers a sample subset of C at one time. It uses the reproduction according to fitness in combination with certain genetic operators to generate new strings generation after generation. In each generation, the genetic algorithm works on a different sample subset of C, the result from the previous generation. This process progressively biases the sampling procedure towards the use of combinations of building blocks with above-average fitness. This means that the genetic algorithm prefers to use strings that contain good building blocks to generate new strings. It has been proven that the number of trials allocated to the observed best schemata (building blocks with high performance fitness) is an exponential function of the number of trials allocated to structures which have less performance fitness [Hol75]. Surprisingly, in a sample subset of size M, the algorithm effectively exploits some multiple of M^3 combinations in exploring C [Hol86], meaning that in each generation, through only M strings being processed, $O(M^3)$ building blocks are involved. Based on the belief that good building blocks constitute good strings, the difference between M and M^3 makes possible a high speedup in the rate of searching C. Holland called this *implicit parallelism*. Moreover, because a genetic algorithm uses the entire sample population as the source to generate new strings, the new strings would be the points in the search space C with different properties. This can reduce the probability of a population converging to a wrong point. In terms of function optimization, the implicit parallelism makes the genetic algorithms capable of handling difficulties, such as local minimum (maximum) and discontinuity [Hol86].

The stochastic process of genetic algorithms is determined by factors such as the size of the sample population, the crossover rate and mutation rate. However, so far there is no concrete rule that can be used to select these factors for a particular system. The so-called genetic disruption problem, good building blocks being disrupted by the genetic operations, is another problem of genetic algorithms. This problem is connected with the crossover and mutation rates. Many studies have shown that it is harmful to the stability of a system to use a high genetic operation rate, especially a high mutation rate.

Premature convergence is another important problem when using genetic algorithms. This problem refers to the situation that all the strings in a population are constructed with similar patterns (schemata) before good strings are found. When this happens, the crossover operator will not add any new string to the population and the mutation operator is usually not strong enough to change the trend of the convergence. The distribution of the fitness in a population determines when and where the population converges. If, at the early stages of a run, the fitness of a few strings in the population is much higher than that of the rest of the population, these a few strings would dominate the selection process and result in a premature convergence. On the other hand, if, at the later stages of a run, the average fitness of the population is close to the best fitness, the selection tends to wander. To over come the above difficulties, fitness scaling was introduced in the context of function optimization. Instead of using the values of the function which is being optimized (hereafter called objective function) as the fitness, a fitness function is used to scale up or down the values of the objective function. Scaling is necessary for keeping appropriate levels of competition throughout a process of optimization. At early stages, the values of the objective function would be scaled back to prevent takeover of the population by some individuals with greater values. At later stages, when the population has largely converged, competition among population members is less strong, the values of the objective function would be scaled up to enlarge differences between population members, so the best ones would still be rewarded. Three kinds of scaling methods have been used: linear scaling [DeJ75], sigma(σ) truncation [For85], and power law scaling [Gil85].

2.2. Genetic-Algorithm-Based Learning (GABL)

The first implementation of a genetic-algorithm-based learning system was Cognitive System Level One (CS-1) reported by Holland and Reitman in 1978

[Gol89a, HoR78]. The system was trained to learn two maze-running tasks. The performance system of CS-1 contains a message list and pool of string rules called *classifiers*. The learning mechanisms used were genetic algorithms and a simple credit assignment algorithm. Since then, many GABL systems have been built (for example, [Gol83, Smi80, Wil86]). These systems can be classified into two frameworks: the Michigan Approach and the Pittsburgh Approach [DeJ88]. In the Pittsburgh Approach, production programs are the objects of genetic operations. The evaluation of "goodness" is associated with the programs instead of the individual rules. Smith's Poker Player is an example of this approach [Gol89a, Smi80]. In the Michigan Approach, each rule is associated with a strength that tells the utility of the rule, and genetic operations are based on individual rules. Systems falling into the Michigan approach are called classifier systems. Since the individuals in a classifier system are single rules (classifiers) instead of complete programs as in the Pittsburgh Approach, classifier systems have much lower computational complexity than the Pittsburgh Approach systems have. Therefore classifier systems can afford to do extensive exploration at lower levels, such as some cognitive process levels. This dissertation concerns genetic-algorithm-based learning models that emphasize low-level processing and discoveries, hence classifier systems are used as the primary foundation.

2.3. Genetic Classifier Systems

Genetic classifier systems are general-purpose, inductive machine learning systems which use genetic algorithms as the discovery mechanism. They learn syntactically simple string rules called classifiers to guide their performances in an environment [Gol89a]. Since CS-1, many systems have been developed in various fields [Gol89a]. A classifier system is a kind of production system, but it differs from conventional production systems in many ways. The most important differences are its low-level representation and processing. Usually, little prior heuristic knowledge about solutions is built into such systems. As a result of adaptation, classifier systems work well with insufficient information and noise.

2.3.1. Description of classifier systems

A classifier system consists of three main components: the classifier and message system, credit apportionment system, and genetic algorithm.

Classifier and Message System

A classifier is a production rule in a string form. It includes two parts: a condition part and an action part. The classifier and message system contains a classifier pool, which is a set of classifiers, and a message list, which is used to hold the messages from either the external environment or the internal classifier activations. Once a classifier is activated, it gets the right to post its action string as a message to the message list, or as an action to the environment. In the latter case, new messages reflecting the changes of the environment are posted to the message list from the environment. In either case, the new messages are used to invoke other classifiers in the next cycle.

In a classifier system defined over the alphabet {'1', '0', '#'}, formally, a condition or an action part of a classifier is defined as:

$$<b_1, b_2, ..., b_k>, b_i \in \{\text{'1'}, \text{'0'}, \text{'\#'}\}.$$

where k is the length of the condition or action string and '#' means either '0' or '1'.

A message is formally defined as

$$<m_1, m_2, m_3, ..., m_k>, m_i \in \{\text{'1'}, \text{'0'}\}.$$

Note that symbol '#' is not used in messages.

Matching messages against the conditions of classifiers is a simple process. Let $m = <m_1, m_2, ..., m_k>$, $m_i \in \{\text{'1'}, \text{'0'}\}$, be a message, $c = <c_1, c_2, ..., c_k>$, $c_i \in \{$'1', '0', '#'}, be a condition of a classifier, $a = <a_1, a_2, ..., a_k>$, $a_i \in$ {'1', '0', '#'}, be the action of the classifier. m and c are said to be matched if the following conditions are satisfied:

1) $m_i = c_i$, if c_i = '0' or c_i = '1'.

2) $m_i = 0$ or m_i = '1', if c_i = '#'.

Message passing is a feature of classifier systems. Messages on the message list would be "passed" through the active classifiers to form new messages. When a classifier matches message m and is chosen to be activated, m will be "passed" through the action of the classifier a to form a new message. The new message $m' = <m'_1, m'_2, ..., m'_k>$ has the values:

1) $m'_i = a_i$ if a_i = '1' or '0'.

2) $m'_i = m_i$ if a_i = '#'.

In brief, if a message satisfies the condition of a classifier and the classifier is chosen to be activated, a new message is generated from the action portion of the classifier by using the '1's and '0's of the action part and passing through the bits of the satisfying message at the '#' positions of the action part. This new message would be either used to change the environment or posted to the message list.

Credit Apportioning System

The credit apportioning system is used to rank or rate individual classifiers according to a classifier's role in achieving reward from the environment. Apportionment of credit is a difficult task faced by most inductive systems. It is easy to determine that a triple jump in checkers is a useful maneuver, but it is much harder to determine which earlier actions made the jump possible [Hol86]. For rule-based systems, the credit apportionment task involves determining which rules in a sequence of active rules have played an important role in determining success. The most prevalent method used in classifier systems is the so-called bucket brigade algorithm [Gol89a].

The bucket brigade algorithm is best described in terms of an economic analogy. The algorithm treats each rule in a reasoning process (a reasoning chain) as a kind of ‘middleman’ in a complex economy. A rule only deals with its ‘suppliers’, the rules sending messages satisfying its conditions, and its ‘consumers’, the rules with conditions satisfied by the messages sent by this rule. Whenever a rule wins a bidding competition, it initiates a transaction in which it pays out part of its strength to its suppliers. As one of the winners of the competition, it becomes active, serving as a supplier to its consumers and receiving payments from them in turn. If a rule receives more from its consumers than it paid out, it has made a profit; that is, its strength increased.

The bucket brigade can be described as follows: when the condition part of a classifier c is satisfied, it makes a bid

$$Bid(c,t) = k \times S(c) \times Strength(c,t)$$

where $Bid(c,t)$ is the bid made by classifier c at time t, $S(c)$ is the specificity, which equals to the number of "non-#"s in the condition part of c divided by the length of the condition, $Strength(c,t)$ is the strength of c at time t, and k is a constant less than 1.

The winning bidders place their messages on the message list and have their strengths reduced by the amount of the bid (they are paying for the right of posting a new message to the message list):

$$Strength(c,t+1) = Strength(c,t) - Bid(c,t)$$

for a winning classifier c. The classifiers $\{c'\}$ that sent the messages matched by this winner have their strengths increased by the amount of part of the bid:

$$Strength(c',t+1) = Strength(c',t) + Bid(c,t) / a$$

where a is the number of the members of $\{c'\}$.

It can be seen that the bucket brigade algorithm assures that stage-setting classifiers receive credit if they make possible later rewarding acts.

Genetic Algorithms in Classifier Systems

Genetic algorithms and credit apportionment algorithms, such as the bucket brigade, constitute the learning component of classifier systems. The role of the genetic algorithms in a classifier system is discovering new classifiers. Assigning strengths to the offspring of genetic operations is another credit assignment related problem in classifier systems that affects the behaviors of the systems. Generally, it is more difficult to deal with the problems of genetic algorithms, such as premature convergence, genetic disruptions, in the context of classifier systems than in function optimization systems since the dynamics of classifier systems are more complicated and the solutions in such systems usually are sets of classifiers with interrelationships. More discussion on this can be found in section 2.4.

2.3.1.1. Match-Activation Cycle Walk-Through

The match-activation cycle here includes matching, conflict resolution and message posting or action activation. Usually, the match-activation is done in the following steps:

1. Initially, environmental messages are put onto the message list.
2. Messages on the message list match against the condition part of each classifier in the population.
3. Conflict resolution is done among all the classifiers that match the same message. Usually, classifiers with higher bids are chosen for message posting or action activation.

4. Old messages are then deleted form the message list. New messages are generated from the action part of the classifiers chosen from the conflict resolution. The new messages would either be posted to the message list or used to change the environment. In the latter case, the changed environment places new messages onto the message list.
5. The strengths of the classifiers are changed according to a credit apportioning algorithm.

2.3.2. Properties of Classifier Systems

Classifier systems differ from the conventional production systems in several important ways.

Genetic classifier systems are low-level computation models which can be used to implement low-level processes such as some cognitive processes. This property allows a classifier system to start with primitive data, in other words, a rich domain theory is not required for this learning method. One application of classifier systems is automating knowledge acquisition, which is a bottleneck in using knowledge-based-system technology. The low-level representation and processing also endow classifier systems with the property of implicit parallelism. Each classifier in the system is an instance of at least 2^l schemata (l is the length of the classifier). Therefore processing on one classifier affects all the schemata of which it is an instance. On the other hand, whether a classifier would be generated is determined by the existence in the population of good building blocks (schemata) which can be used to construct the classifier.

One of the main obstacles to learning in a conventional symbolic model has been complex rule syntax. In such a system, generating new rules involves grammatical constructions and it is hard to do the matching efficiently. To accelerate the matching process, the Rete match algorithm was proposed and used in symbolic production systems OPS interpreters [For82, Jac86]. To use this algorithm, the left hand sides of the rule base must be compiled into a tree-structured network. The initial compilation, and its update later on in a changing environment, are not trivial. The algorithm avoids many redundancies in the matching process, but the speed is still of $O(n)$, assuming n is the number of the production rules being matched. Classifier systems overcome the above problems by using basic representation primitives, '0', '1', '#', and rule strings over these primitives. In a binary string coded classifier system, it is trivial to build an ordered binary tree of the left hand sides of the classifiers and the update of such a tree is not so costly. In this case, the match speed is of $O(\log_2 n)$, where n is the size of the classifier pool. The tree structure is used in our experiments which are discussed in later chapters.

Classifier systems have not only the implicit parallel search property but also the explicit parallelism. Large numbers of classifiers can be active at the same time. By doing this, classifier systems permit multiple activities to be coordinated simultaneously. Decision making is postponed until a conflict resolution is necessary. This usually happens in the following two circumstances. First, the candidate actions are mutually exclusive. Decisions must be made on which actions would be carried out. Second, the number of new messages is greater than the size of the message list. In this case, decisions must be made on which messages would be posted onto the message list.

In classifier systems, rules generated by the inductive mechanisms need not be universally correct because of the competition and the way in which a new classifier is generated. Therefore, the computationally overwhelming requirements for global consistency of rules are avoided. For example, in a default hierarchy (this concept will be explained later), specific rules, exceptions of the general rules, coexist with the general rules. In the competition process, the specific rules would usually win. This example shows that although the general rules may not be completely correct if examined, the entire rule set can still be correct based on a good competition balance between the general and specific rules. This illustrates the fact that it is not necessary in a classifier system that every rule is completely correct.

2.4. Problems with Classifier Systems

Although classifier systems show the promising properties, their applications are troubled by a number of problems. Compared with genetic algorithm function optimization systems, where the solution usually contains only one (or a few) string(s), and the information about the utility of a string can be obtained from the objective function, classifier systems solve problems where the solution(s) consist of many classifiers and it is hard to evaluate the utility of a classifier at each step precisely. Maintaining good classifiers and classifier structures, while introducing new useful classifiers into the system, has been a problem. This usually results in difficulties in generating and maintaining all the sub-solutions and necessary rule relationships (such as default hierarchies) in a population. Often the result is poor and unstable system performance. This is reflected by the fundamental problems of rule clustering and rule association.

2.4.1. Rule Clustering Problem

The rule clustering problem occurs because it is possible that a string with high strength (utility) becomes a super-pattern over the population [Gol89a]. Since strings with high strengths are likely to be chosen as objects for genetic operations, gradually the whole population will converge to strings with similar patterns. In other words, the result state clusters to some sub-solutions. This problem can happen in two situations:

when the result state includes more than one independent classifier as part of the solution, and when several related classifiers constitute one solution.

A similar problem has been pointed out with genetic algorithms being used for function optimization [DeJ75]. This problem is more serious when genetic algorithms are used in learning systems. Usually the goal states of a learning are a set of related rules, which means, to be effective, the system must find *all* the rules and the relationships among the rules. Also, in classifier systems, the difficulty in determining correctly the strengths of the offspring of genetic operations makes it harder to remove "bad" classifiers generated by genetic operations.

DeJong has proposed a method called *crowding* to limit the number of strings for each sub-solution in the population [DeJ75]. Instead of deleting individuals at random, a small subset of the population is randomly selected. The individual in that subset most similar to the new individual is then chosen for deletion. After a certain point, new individuals begin to replace their own kind and the proliferation of a species (sub-population) is limited. DeJong experimentally showed that system performance improves with crowding. The application of this method in classifier systems involves the questions of how to define the similarity between the individuals and how to choose the size of the subsets for comparison. The answers to the above questions are important, especially when the patterns of the sub-solutions are similar.

Booker proposed a method called *restricted mating strategy* to solve the rule clustering problem which includes the problem of sub-solutions' competing with each other [Boo82]. In *restricted mating*, only those strings that belong to the same example patterns will be allowed to conduct genetic operations with each other. The example pattern here refers to the examples of each sub-solution or some characteristic description of each sub-solution. Booker used this strategy in his GOPHER-1 system

[Boo89]. In GOPHER-1, classifiers that are excited by the same message are called a classifier cluster. The genetic operations are restricted to classifiers in the same cluster. This reduced the clustering problem in the sense that, globally, classifiers for different messages would not disrupt each other. However, locally, in each classifier cluster, premature convergence could be a serious problem when the search space is much larger than the sample space.

2.4.2. Rule Association Problem

The other major difficulty with classifier systems is the rule association problem, achieving cooperation among a set of rules. Cooperation includes rule chains and rule default hierarchies. In the rule chain situation, a set of rules form a reasoning chain. Chaining is considered an important strength of classifier systems. Default hierarchies refer to rule hierarchies ordered by subordinate or superordinate relations [HHN86]. Rules that are more general are derived from conclusions based on incomplete information about a world. Therefore these general rules contain some default hypotheses about the world. However, these hypotheses or expectations can be overridden by more specific evidence introduced later on. Hence the more specific rules based on more knowledge about the world can complement the general ones. Thus the general and specific rules cooperate to model the world correctly. Since the rule population contains only a small part of the entire possible rule space, the problems in rule association are how to generate and maintain the rule associations.

Rule clusters, rule chaining and default hierarchies are important phenomena in genetic-algorithm-based systems. They are closely connected with the concept of solution formation, category formation, rule (concept) generalization, and specialization. The common characteristic of these phenomena is rule association. The behavior of the rule association affects the robustness and the stability of genetic-algorithm-based systems. However, conventional classifier systems lack effective mechanisms for forming and maintaining rule associations. For example, several efforts have been made to solve this problem in conjunction with default hierarchies. One way to form and maintain default hierarchies is to favor more specific classifiers when bidding to fire classifiers [Rio89]. However this scheme encourages the survival of more specific classifiers over more general ones. For example, "#000/0" and "#100/0" would survive over "##00/0", or, even worse, classifiers with no '#'s, such as "0000/0", "0100/0", "1000/0" and "1100/0" would survive over the classifiers with '#'s. This phenomenon is described as ***starving the generals*** [Rio89, Wil88]. The recent scheme of ***necessity auction and separate priority factor*** has not solved the above problem either [SmG90]. The difficulty here lies in the lack of a strong relationship among the exception classifiers and default classifiers. Therefore, there is no distinction between a real exception classifier and a more specific classifier that is covered by the default classifier.

2.4.3. Initial Populations

Genetic-algorithm-based classifier systems start with an initial population of classifiers that evolve from one generation to the next. Over time, feedback on the "goodness" of the system output is used to decide which classifiers should be maintained in the population and used as progenitors for the next generation. Eventually, the system may converge on a solution set of classifiers. The initial population serves as the "background knowledge". It influences the common genetic algorithm problems of premature convergence and over-dominance of a few classifiers. A bad initial population tends to either converge fast at the beginning before the correct solutions are generated, or take a long time to generate the correct solution(s).

The main reason that different initial populations may give different results lies in the types and number of the schemata contained in the population. In genetic-algorithm-based learning, the search is based on the schemata rather than individual classifiers. This phenomenon is called ***implicit parallelism*** by Holland [Hol86]. The more schemata contained in a population, the higher the degree of implicit parallelism. Given that the mutation rate is usually low in a system, crossover is the main operator that creates new schemata in genetic-algorithm-based learning, so the following discussion about locating or generating useful schemata will be based on this operator. If the initial population contains few different schemata (i.e. if many schemata duplications exist), then the population tends to converge quickly to a wrong classification. This means that the population reaches a point where any crossover operation would not bring in new classifiers, but the population has not converged to a correct solution.

Another issue is that the schemata contained in the initial population may be very different from the schemata in the solution. In this case, the system will usually take a longer time to generate the correct schemata than if the schemata contained in the initial population are similar to the ones sought. For instance, to generate a schema that contains four '#'s from schemata having only one '#', at least two generations of crossover are needed. After the first crossover, a schemata containing two '#'s might be generated. Another crossover on the two '#'s schemata may introduce a schema with four '#'s. However, if we start with a two '#'s schemata, one with four '#'s could be introduced with a single crossover operation. Conversely, if the majority of schemata in an initial population have five '#'s, it may take several crossover operations to create one with a single '#'. If the initial population is biased to some schemata that are not similar to the correct ones, it may take a long time to overcome these biases. In other words, similarity and diversity of schemata are important aspects that affect performance and should be a consideration in selecting an initial population.

The importance of a good initial population has not been adequately studied in the literature. The initial population is usually chosen randomly. Our research suggests this may be a dangerous practice. Experimental results demonstrate the expected benefits achievable from the selection of a good starting population. Our studies on the initial population are presented in chapter 3.

2.4.4. Remarks on the Issue of the Problems in Genetic Learning

While discussing the problems of rule clustering and rule association in 2.4.1 and 2.4.2, we have pointed out that these two problems are associated with the difficulties in credit apportionment. We would like to elaborate this issue here.

In genetic classifier systems, the credit apportionment is required in two situations. One is when a decision about which classifier(s) are responsible for success or failure is made. This is usually done by the bucket brigade algorithm. However, its effectiveness is restricted by the limitations of the algorithm [WiG89]. The other situation is when decisions about the strengths of new classifiers generated by the genetic operations are made. This is a more difficult problem in classifier systems than in function optimization systems, since usually there is neither local nor global information available for the evaluation of new classifiers before they experience trials. Currently, assigning strengths to new classifiers is done in an ad hoc way. For example, assigning the average strengths of the parents to their children or the average strengths of the population to new classifiers are used.

The distribution of the strengths determines where the population evolves. Particularly, it determines if desired classifiers would stay in the population and produce offspring, and the undesired ones be removed from the population. Since the cooperation among classifiers is not strongly supported by the data structures in classifier systems, the performance of the credit assignments becomes vital to the resolution of the problems of rule clustering and rule association. To reduce the dependence of the resolution on the credit assignments, our work pays attention to the data structures in a system.

2.5. Comparing GABL with other Machine Learning Paradigms

Learning ability is an important aspect of intelligence. Machine learning is a way to simulate and understand the mechanism of human learning and also a powerful tool for artificial intelligence systems to deal with the real world. In the real, complex, unpredictable world, it is impossible to describe everything in advance. Thus learning ability is needed so that lack of knowledge, or incomplete knowledge, can be compensated by the processing of current information [Tsy73]. Over the years, research in machine learning has been pursued with varying degrees of intensity, using different approaches and placing emphasis on different aspects and goals.

Many issues are involved in determining the characteristics of a learning system. Among them, the representation issue and the mechanisms of generating new knowledge are the most important factors that affect the behavior of the system. Different emphases can result in different ways of classification. Since different approaches share some common aspects and there is no formal definition for the approaches, an exclusive classification of machine learning paradigms has not been a trivial task. Machine learning has been classified into *symbolic learning*, *genetic learning* and *neural net learning* [Mic86]. A more recent classification is *inductive learning*, *analytic learning*, *genetic learning* and *neural net learning* [Car90].

Depending on how *inductive learning* is interpreted, one can see that *genetic* *learning* and *neural net learning* could be two sub-paradigms of an *inductive learning*. In our discussions, the following interpretations are used. *Inductive learning* here refers to the conventional inductive concept learning systems. In these systems, a (or multiple) concept description(s) is (are) induced from a sequence of positive and negative instances. ID3 is an example of this type of learning [Qui83]. In ID3, instances of a known class are described in terms of a fixed collection of attributes. A classification rule in the form of a decision tree is constructed according to the values of the attributes of the instances. A recent development of ID3 is C4, in which probabilities are used to deal with noise and unknown attribute values [Qui90]. This decision tree method requires that the attributes used are the key factors that can classify the objects into the classes. The order in which the attributes are tested is also important for the success of the method.

In *analytic learning*, built-in analysis strategies are used to generate new knowledge, such as rules, through domain knowledge and past problem solving experience. The compiling mechanism in ACT* [And86], and chunking mechanism in SOAR [LRN86] are examples of such strategies that are used to generate new rules. One of the compiling strategies in ACT* is the composition strategy, in which whenever a sequence of production rules achieve a goal, a single rule would be formed which will have the same effect as the set of the rules. In SOAR, the problem solving is considered as a search through a problem space for the states that can achieve the goal criteria. When an impasse occurs in this process, a sub-goal is generated and the same search process is used to achieve this sub-goal. When an impasse is resolved, the chunking mechanism will generate a new rule or a set of rules that can be used to avoid the same impasse in the future.

Although the analytic and inductive approaches have fundamental differences between each other, a common ground shared by these two paradigms is that new knowledge is constructed at the concept level, meaning that the processing is conducted on the symbols that represent domain concepts. This fundamental characteristic distinguishes these two paradigms from the genetic learning and neural net learning. Therefore, we refer, hereafter, the analytic and inductive paradigms as concept level symbolic approach.

A neural network system consists of a set of units and links connecting the units. Each unit is associated with an activation value and activation function. Each link is associated with a strength. Usually a neural network has three kinds of units: input units, "hidden units" and output units. By turning certain units on or off according to an activation function, the systems respond to the input from the external world with the output units [RHM86].

The dynamics of the network and the learning mechanism for adjusting the strengths of the links are two aspects of neural networks. The former is mainly controlled by the activation function, while the latter is an algorithm for changing the strengths of the connections so as to minimize the differences between the actual output and the expected output during the runs of the systems.

The use of the analytic strategies requires rich domain knowledge. Therefore, this type of systems usually contain numerous built-in concepts, knowledge structures and domain specific constraints.

As described in previous sections, genetic classifier systems consist of production rule-like classifiers. This representation is similar to that of symbolic production rule systems. However, the genetic discovery is conducted at low level as opposed to the concept level. As well, the way in which new rules are generated distinguishes genetic classifier systems from the other paradigms.

It is easier to see the differences between the symbolic approach and neural and genetic approaches than to tell the differences between the neural networks and genetic classifier systems.

The symbolic approach has richer, clearer representation structures and is amenable to representing higher level and more complex knowledge. It is easier for a human to communicate with such systems, whereas the genetic and neural approaches are less articulate. Especially in neural network systems, the entire network is an internal model which reflects the underlying dynamics of its environment. A single unit (node) does not have an independent meaning. Nevertheless, the low-level representations in genetic classifier systems and neural networks bring these systems several important features. Massive parallelism is a distinct feature of these systems. In such systems, each element (unit in neural networks and classifier in classifier systems) is highly interconnected with many other elements and the processing to each element is local. This provides a base for parallel processing. As discussed in previous sections, massive implicit parallelism is another important feature of classifier systems.

Another feature of neural networks and classifier systems is their capability for describing cognitive processes. Many cognitive phenomena occur below the level represented by concept level symbols [BeF88].

Genetic classifier systems and neural networks are both low level computational models. But they use different learning mechanisms and representation frameworks. The adaptive learning ability of a genetic classifier system is determined by two learning algorithms, a credit apportionment algorithm, such as the bucket brigade, for adjusting the strengths and a genetic algorithm for discovering new classifiers. There is only one learning algorithm in a neural network system. More research needs to be done to conclude the differences stemming from the different learning algorithms.

The dissimilarity of genetic classifier systems and neural network systems can also be seen from representation view points. Classifier systems use production rule-like classifiers representation. Each classifier has an independent meaning. Techniques used in symbolic learning such as building knowledge structures, using variables, can be naturally adopted in classifier systems [ShS89], whereas it is difficult to adopt these techniques in a neural network system [Nor86].

Before we go into the summary part, we use the following examples to illustrate some differences between symbolic learning approaches and genetic learning.

The first example shows the differences between genetic learning and the decision tree method which belongs to the symbolic inductive learning paradigm. Consider learning the concept of "rectangle". In the decision tree method, the learning could be conducted with the following attributes:

<*length−relation* , *length−relation* ,*relation* , *angle* , *angle* , *angle* , *angle* >,

where *length-relation* is the relation between the length of the two opposite sides and its possible values are EQUAL or NOT-EQUAL, *relation* is the relationship between the two sets of the opposite sides and its possible values are EQUAL and NOT-EQUAL, *angle* is the value of each angle. A set of instances could be the following:

<*EQUAL* , *EQUAL* , *EQUAL* , 90, 90, 90, 90> → *Non−Rectangle* ,
<*EQUAL* , *EQUAL* , *NOT−EQUAL* , 90, 90, 90, 90> → *Rectangle.*

Based on the value of the attributes, a decision tree can be built on a set of instances to classify the items into different categories. In this example, the attributes *length-relation* and *relation* are vital to the success of this method. It will not work, if instead, the attributes are the actual lengths of the four sides.

Using the genetic method, learning can be conducted at a lower level. That is, the concept of the relationship between the sides is not necessary. If the following attributes are used:

$$<length, length, length, length, angle, angle, angle, angle>,$$

where ***length*** is the length of a side and ***angle*** is the angle between two sides, the following rule could be learned:

$$v_1\ v_1\ v_2\ v_2\ 90\ 90\ 90\ 90 \rightarrow Rectangle.$$

The condition part of the rule can be interpreted as:

1. The length of the first pair of opposite sides is v_1.
2. The length of the second pair of opposite sides is v_2.
3. v_1 is not equal to v_2.
4. All the angles are 90 degrees.

Another example shows the difference between the generalization and discrimination mechanisms in ACT* and the mechanisms in a genetic classifier system. Consider the following example in language acquisition. In ACT*, if there are two production rules:

1. If the goal is to generate the plural form of APPLE,
   Then say APPLE + s.
2. If the goal is to generate the plural form of CAP,
   Then say CAP + s.

The generalization mechanism would try to generate a new rule that would cover the above two rules. The new rule would be

3. If the goal is to generate the plural form of X,
   Then say X + s.

When found out that nouns ending with 's' do not comply with Rule 3, the discrimination mechanism would generate the following two rules:

If the goal is to generate the plural form of X
and X is not ended with ‘s’,
Then say X + s.

If the goal is to generate the plural form of X
and X is ended with ‘s’,
Then say X + es.

We notice here, in the application of the generalization and discrimination strategies, the conscious separation of the last letter from the rest of a word that is being considered is important. The system must be told what to look at to generalize or discriminate rules.

Consider the variable genetic learning approach [ShS89]. To simplify the representation, we use strings over the 26 English letters plus ‘*’ to represent a noun. Each letter in a noun is represented by two alphabets in the way that "AA" represents ‘A’, "BB" represents ‘B’ and so on. A ‘*’ combined with another letter represents a variable. For example, "A*" and "*C" are two variables. Thus, word "apple" would be represented as AAPPPPLLEE and A*B*C*D*SS reads as "any five-letter word ended with a ‘s’". To have more than one variable, at least two alphabets are needed. For instance, A* and B* are two variables. This is the reason that two alphabets are used to represent one letter in a word. The following set of rules can be considered as examples of classifiers.

1. AAPPPPLLEE / AAPPPPLLEESS
2. TTAABBLLEE / TTAABBLLEESS
3. A*B*C*D*SS / A*B*C*D*SSEESS

where classifier 1 says that the plural form of "apple" is "apples", classifier 2 says that the plural form of "table" is "tables" and classifier 3 can be interpreted as "the plural form for any five-letter word ended with a ‘s’ is the word plus "es"". Through the genetic operations, the following two classifiers may emerge:

A*B*C*D*E* -> A*B*C*D*E*SS,
A*B*C*D*SS -> A*B*C*D*E*EESS.

The first classifier is equivalent to

> If the goal is to generate the plural form of a five-letter word WORD,
> Then say WORD + s.

The second classifier is equivalent to

> If the goal is to generate the plural form of a five-letter word WORD
> which is ended with a ‘s’,
> Then say WORD + es.

Using genetic classifier system approach, a system does not need to be told what to look at for generalization and discrimination. This is because that the more general classifiers are not derived directly from the more specific classifiers. Under the computation mechanisms, all the classifiers are the same in the sense that each of them is an individual of the population. Hence, in such systems, there is no need to separate consciously the last letter of a word from the rest.

This example is used to illustrate the differences between the paradigms. The task in this example is not easy for either of the paradigms. Depend on what the emphasis is, discovering the regularity from some primitive data or extracting the regularity based on the domain theory, each paradigm has its advantages over the other.

Although there are fundamental differences between the paradigms, they share some common issues. Credit assignment problem is shared by the inductive models, genetic learning models and the inductive mechanisms used in the analytic systems. Credit assignment algorithms based on different theories, such as probability models and economics models, have been used with different emphases.

From functionality point of view, the symbolic learning has the advantage of representing high-level knowledge and knowledge structures. Since the control strategies are applied at the concept level, systems of this paradigm can process efficiently at high-level and it is easier to explain the behaviors of such systems. But it lacks the ability of revealing low-level characteristics. Genetic classifier systems use the symbolic level structure (production rules) with low-level representation primitives (bits) and have both symbolic level (rule matching) and low-level (genetic algorithms on schemata) processing while neural net systems use low-level representation and process at a low-level only. However, the genetic and neural net learning methods lack the mechanisms of taking advantages of heuristic knowledge.

We would like to make it clear that the '0', '1' or any other alphabets representation is not the fundamental characteristic that distinguishes genetic classifier systems from the other paradigms. The fundamental difference is the learning mechanisms. The reason that the symbolic approach is so called is not that the knowledge is represented by symbols. It is rather because that the processing in such systems are based on the symbols that represent domain concepts. The word "low-level" has slightly different meanings in different situations in our discussions. When referring to representations, it means the representations are constructed from a set of alphabets and each individual alphabet usually does not represent a concept in the domain. When referring to processings, it means that the processings are conducted at a level that is below the given concepts.

In summary, each paradigm has advantages over the others in certain situations. When the solution of a problem does not go beyond the built-in analysis strategies, symbolic approach may be more efficient than the other methods. When the domain theory is not rich enough to apply the general strategies, genetic and neural net learning approaches may be more effective. The fundamental characteristic that distinguishes the paradigms from each other is the mechanisms of generating new knowledge.

Although there exist many machine learning models, the above difference classifies those models into the paradigms we discussed. Since the purpose of this dissertation is not to make a detailed comparison between all existing methods, the above discussions did not go into the sub-paradigms of each class.

Chapter 3

Impact of Initial Populations on Performance

The importance of initial populations has been discussed in chapter 2. In this chapter, analyses are presented on issues related to the quality of an initial population. Two measures are defined and used to guide the selection of initial populations. New algorithms are presented for generating better initial populations to maximize the potential of a classifier system. Experiments comparing the performances of randomly generated initial populations and initial populations generated by the new algorithms were conducted. Both the theoretical analyses and experimental results show that the problems of premature convergence and over-dominance of sub-solutions are significantly reduced by a good choice of an initial population. Experimental results demonstrate the expected improvements achievable from the selection of a good starting population.

The system about which we are going to discuss is a four-tuple $<E, C, G, K>$, where E is the environment, C is a set of classifiers, G is a set of categories and K is the prior heuristic knowledge about the solutions:

$E = \{e_i\}$, where e_i $(i = 1, 2, ..., n)$ is an environment state,

$C = \{c_i\}$, where c_i $(i = 1, 2, ..., m)$ is a classifier, and

$G = \{g_i\}$, where g_i $(i = 1, 2, ..., j)$ is a category.

The classifiers are strings of l characters over the alphabet {'0', '1', '#'} †. In the following discussion, we assume that K is ***null***. That is, little knowledge that is beyond the description of the problem is available. Also, for our convenience, the length of a string refers to the length of the condition part of a classifier.

---

† The results generalize to an arbitrary alphabet. For simplicity, the alphabet is assumed to contain 3 characters, as is usual in classifier systems.

3.1. Examining the Quality of an Initial Population

There have been no formal measurements for characterizing an initial population. This makes it difficult to study the initial population problem quantitatively. Two new measures are introduced here for examining the quality of an initial population. Consider all the classifiers contained in a population P as a *population array* of m rows (one for each classifier) and l columns (one for each symbol in a classifier). Let $AVG = m / c$, with c being the number of the symbols used in the system (usually c is 3 for classifier systems). Let $a_{i,j}$ be the number of times that symbol j appears in column i, $maximum_i = \max a_{i,j}$, and $minimum_i = \min a_{i,j}$, for $j = 1, 2, ..., c$.

Definition 1: *EvenDiversity*$(P, i) \equiv ED(P, i) \equiv AVG - (maximum_i - minimum_i)$, *where P is a population, i is the ith column of the population array.*

Example: Let the population array be

$$P_0 = \begin{bmatrix} 0 & 0 & \# & 0 \\ \# & 1 & 0 & 0 \\ 0 & \# & \# & 0 \end{bmatrix}.$$

In this case, $m = 3$ and $c = 3$, so $AVG = 3 / 3 = 1$. '0' appears twice in column 1, '1' does not appear in column 1, and '#' appears once in column 1. So, $maximum_1 = 2$, and $minimum_1 = 0$. $ED(P_0, 1) = 1 - (2 - 0) = -1$. Similarly, we have $ED(P_0, 2) = 1$, $ED(P_0, 3) = -1$, and $ED(P_0, 4) = -2$.

Lemma 1: *If the number of the times that each symbol appears in a column are not the same, then* $ED(P, i) < AVG$.

Proof: If the number of times that each symbol appears in a column are not the same, then $\Delta = maximum_i - minimum_i > 0$. Therefore, $ED(P, i) = AVG - \Delta < AVG$. $\square$

Lemma 2: *The maximum value of* $ED(P, i)$ *is* AVG. This value is achieved *when all the symbols appear evenly in the column.*

Proof: Follows immediately from the definition of ED and Lemma 1. $\square$

Definition 2: $EvenDiversityPopulation(P, l) \equiv EDP(P, l) \equiv \sum_{i=1}^{l} ED(P, i) \,/\, l$, *where P is the population and l is the length of the string.*

In the above example,

$$\begin{aligned} EDP(P_0, 4) &= \sum_{j=1}^{4} ED(P_0, j) \,/\, 4 \\ &= (-1 + 1 + (-1) + (-2)) \,/\, 4 \\ &= -3/4. \end{aligned}$$

By the definition, the maximum value of $EDP(P, l)$ over P of the same population size is AVG.

The order of a schema is defined to be the number of positions with fixed values in this schema. For example, schema "00**" has two positions with fixed values. Both position 1 and position 2 (from left to right) has value ‘0’. Therefore it is an order 2 schema. Schema "**#*" has its third position with value ‘#’, so it is an order 1 schema (the ‘*’ represents doesn’t matter, here it could be a ‘0’, ‘1’ or ‘#’, therefore its value is not considered as fixed). By the definitions, we can see that the larger the $EDP(P, l)$ is, the more order 1 schemata are represented in population P and the more even the distribution of the order 1 schemata is. Using the example above, it can easily be demonstrated why having a variety of different order 1 schema in an initial population P_0 is important. In P_0, ‘#’ does not appear in column 4, which means schema "***#" is not contained in the population. Similarly, we can find many other schemata, such as "***1" and "1***", that are also not part of the population. If any of these schemata are part of the solution, it may take a long time to generate them using mutation since the mutation rate is usually very low to maintain the stability of a system. Even with the *create* operator [Wil86] or *cover detector* [Rio89], it is not easy to introduce proper schemata that contain '#'s. Another point illustrated by P_0 is that column 1 contains two '0's, but only one '#'. This means that a bias (possibly unwanted) is imposed on the population initially (for example, '0' has a higher chance of survival in column 1).

Lemma 3: *If the number of the times that each symbol appears in a column are not the same,* $EDP(P, l) < AVG$.

Proof: This follows from the definition of $EDP(P, l)$, Lemma 1 and Lemma 2. □

Theorem 1: *When* $EDP(P, l) = AVG$, *the number of order 1 schemata reaches its maximum for population P.*

Proof: Suppose that the number of order 1 schemata contained in the population is not the maximum number of order 1 schemata that could be contained in the population. Then there must be an unnecessary order 1 schemata duplication. Without loss of generality, we assume that the unnecessary duplication is two "#***"s. So, we change one "#***" to "1***" (or "0***"). If this change leads to one more new schema in the population, "1***" must not have existed in the population before. By Lemma 3, we have that $EDP(P, l) < AVG$. This contradicts the premise that $EDP(P, l) = AVG$. Thus the number of order 1 schemata in the population is the maximum number of order 1 schemata that the population could contain. □

The number of generations of the crossover operation that are needed to create a correct solution from the initial population is another parameter that affects performance. There are primarily two factors that affect this number. The first, called *physical factor*, refers to the "physical condition" of the population. It is the number of crossover operations needed to construct a solution string from the population assuming that we always pick the most suitable strings to crossover. The second factor includes the influences affecting the length of time needed to select proper strings, and is called *dynamic factor*. It is obvious that the fewer the number of crossovers needed, the better the performance of the system in both stability and efficiency. A new measure is proposed to quantify how far a population is away from the solution.

Definition 3: *NonSimilarity* $(S, P) \equiv NS(S, P)$ *is the minimum number of generations of the crossover operation that are needed to get solution S from population P. If P has converged to a wrong solution,* $NS(S, P)$ *is* ∞.

From the definition, $NS(S, P)$ is a measure of the distance between a population and a solution. When $NS(S, P) = \infty$, it is impossible to reach S from P, assuming no mutation operations are carried out †.

Theorem 2: *If* $EDP(P, l) = AVG = PopulationSize / 3$, $NS(S, P)$ *is not* ∞ *for any string S over {'0', '1', '#'}. In other words, any string can be generated by several applications of the crossover operator on proper strings in population P, assuming that the system can keep the useful order 1 schemata around long enough to be selected.*

Proof: This can be proved by constructing an arbitrary string S from population P. Without loss of generality, assume $S =$ "#001". Since $EDP(P, l) = AVG$, every column of the population array contains all three symbols ('0', '1' and '#'), which means that schemata "#***", "*0**", "**0*" and "***1" are in the population. Therefore, by crossing over on strings that contain the above schemata, we can get string

---

† Even with mutations, it usually takes a very long time to overcome the trend of converging to the wrong solution.

"#001". □

From the above discussion, a good initial population should have as many different schemata as possible and, if some duplications are not avoidable, they should be evenly distributed in different schemata to avoid biases. For example, an initial population of size 6 that contains each of "#***", "0***" and "1***" twice is better than an initial population that contains four "#***"'s and one each of "0***" and "1***". When the number of times each of '0', '1' and '#' appears in any column of the population array are the same, by Theorems 1 and 2 we may have a good initial population that contains a variety of schemata and is not biased. This is a characteristic of an initial population that is not vulnerable to the so-called premature convergence problem.

Next, desirable properties of the rows of an initial population are discussed. As mentioned previously, $NS(S, P)$ can be considered as the distance between a solution S and population P. To improve performance, we need to minimize $NS(S, P)$. In the following, a way of reducing $NS(S, P)$ probabilistically is presented.

In a classifier system, the probability that a string is a solution or part of a solution is not the same for all the strings. A string S is included by $2^{(l-i)}$ strings, where l is the length of the strings and i is the number of '#'s in string S. This means that if any of the $2^{(l-i)}$ strings is included in a solution set, string S is part of the solution. This suggests that the more '#'s a string has, the less probability that the string is part of a solution. An extreme example is that the chance that "######/1" is a solution or part of a solution is much smaller than that of "000000/1". Hence, a string without any '#' would have the largest probability of being part of a solution. However, we want strings that are more general and that can reveal some relationships between individual cases. Besides, for problems with infinite number of individual cases, it is impossible to solve the problems without looking at the relationships among the individual cases. The above analyses suggest that strings that are neither too general nor too specific should be used to minimize the distance between the initial population and the solution set.

The desirable properties of the rows can also be seen from the view point of schemata distribution.

3.2. New Algorithms for Generating Initial Populations

The above analysis indicates that diversity across each row and each column of the population matrix is desirable. Based on this, new algorithms are presented that can be used to generate an initial population with a high degree of diversity for genetic-algorithm-based learning.

By *Theorem 1* in section 3.1, when symbols '0', '1' and '#' are evenly distributed in each column, the population contains all the order 1 schemata. This means the population contains all the "basic information" and hence there is no bias at the beginning. Figure 3.1 shows the code for doing this in a C-like notation.

Discussions on $NS(S, P)$ in section 3.1 indicates that a moderate number of '#'s in a row is desired to minimize the distance between an initial population and the solution set. Figure 3.2 presents the code for imposing both column and row controls in a C-like notation.

The above algorithms are actually two groups of algorithms. The column controlled algorithms reflect the idea of including a variety of allele and giving no bias to any of them in an initial population. The column & row controlled algorithms do more

```
ColumnControl( )
{
    /* Generate the classifiers */
    for( i = 0; i < PopulationSize; i = i + 1 )
    {
        /* Generate the condition part */
        for( j = 0; j < StringLength; j = j + 1 )
        {
            /* Generate a symbol using column         */
            /* control. The more a symbol has appeared */
            /* in a column, the less chance that      */
            /* the symbol should be generated again.  */
                count the number of times each symbol
                appears in column j
            probability of generating a symbol is in
              inverse proportion to its past frequency
                of occurrence in column j
            generate symbol using these probabilities
        }
        /* Generate the action part similarly */
    }
}
```

Figure 3.1: Generating a Column Controlled Population.

in generating an initial population that is closer to the solution set. The variants of the above two algorithms, shown in Figures 3.3 and 3.4, were used in our experiments. The difference between ColumnControl-1 and ColumnRowControl-1 is that the former uses the number of the appearances of each symbol in the column to guide the generation of next symbol, whereas the latter uses the total number that a symbol appears in both the column and the row.

3.3. Experiments

Learning Boolean functions was used as the tasks in the experiments. Our goal is to show how a good initial population can reduce problems of premature convergence and over-dominance of sub-solutions, and improve performance. Two kinds of

```
ColumnRowControl( )
{
    /* Generate the classifiers */
    for( i = 0; i < PopulationSize; i = i + 1 )
    {
        /* Generate the condition part */
        for( j = 0; j < StringLength; j = j + 1 )
        {
            /* Generate a symbol using row and column  */
            /* control. The more a symbol has appeared */
            /* in a column/row, the less chance that   */
            /* the symbol should be generated again.   */
                count the number of times each symbol
                appears in column j
            count the number of times each symbol
                appears in row i
            probability of generating a symbol is set
                to the inverse of its past frequency
                of occurrence in row i and column j
            generate symbol using these probabilities
        }
        /* Generate the action part similarly */
    }
}
```

Figure 3.2: Generating a Column and Row Controlled Population.

functions are selected. The first kind is functions whose solution set consists of many different schemata. The second kind is functions whose solution set contain only a few different schemata. For instance, the classifier set

```
{    001##0/0,
     #1011#/1,
     1##001/0   }
```

contains various schemata in different classifiers since there are very few duplications of schemata in the population. In contrast, the classifier set

```
{    00####/0,
     1#0###/0,
     110000/1   }
```

contains fewer schemata because of the duplications, such as the repetition of schema

```
ColumnControl-1( )
{
    /* Generate the classifiers */
    for( i = 0; i < PopulationSize; i = i + 1 )
    {
        /* Generate the condition part */
        for( j = 0; j < StringLength; j = j + 1 )
        {
            /* Generate a bit */
            a[ 0 ] = PopulationSize - NumbTimes( '0', j )
            a[ 1 ] = PopulationSize - NumbTimes( '1', j )
            a[ 2 ] = PopulationSize - NumbTimes( '#', j )
            a = a[ 0 ] + a[ 1 ] + a[ 2 ]
            randomly generate a number 'rand' in range 0 ... a−1
            if( 0 ≤ rand < a[ 0 ] )
                generate a '0'
                else if( a[ 0 ] ≤ rand < a[ 0 ] + a[ 1 ] )
                generate a '1'
            else
                generate a '#'
        }

        /* Generate the action part similarly */
    }
}

NumbTimes( character, which )
{
    return the number of times 'character' appears in 'which' column
}
```

Figure 3.3: Generating an Initial Population with Controls on Columns.

"***###" in the condition parts of the first and second strings and schema "1*****" in the second and third strings. Our results given later have shown that improvements achieved by using a good initial population are more obvious for the first kind functions than for the ones of the second kind.

The functions of the first category that we experimented with include the following:

$f_1(x_0, x_1, x_2, x_3, x_4, x_5) =$

```
ColumnRowControl-1( )
{
        /* Generate the classifiers */
        for( i = 0; i < PopulationSize; i = i + 1 )
        {
                /* Generate the condition part */
                for( j = 0; j < StringLength; j = j + 1 )
                {
                        /* Generate a symbol using row and column  */
                        /* control. The more a symbol has appeared */
                        /* in a column/row, the less chance that   */
                        /* the symbol should be generated again.   */
                        count the number of times each symbol
                                appears in column j
                        count the number of times each symbol
                                appears in row i
                        probability of generating a symbol is set
                                to the inverse of its past frequency
                                of occurrence
                        generate symbol using these probabilities
                }
                /* Generate the action part similarly */
        }
}
```

Figure 3.4: Generating an Initial Population with Controls on Columns & Rows.

$(NOT(x_0)\ AND\ NOT(x_1)\ AND\ NOT(x_2)\ AND\ x_3)$

$OR\ (NOT(x_0)\ AND\ NOT(x_1)\ AND\ x_2\ AND\ NOT(x_3))$

$OR\ (NOT(x_0)\ AND\ x_1\ AND\ NOT(x_2)\ AND\ NOT(x_3))$

$OR\ (NOT(x_0)\ AND\ x_1\ AND\ x_2\ AND\ x_3)$

$OR\ (x_0\ AND\ NOT(x_1)\ AND\ NOT(x_2)\ AND\ NOT(x_3))$

$OR\ (x_0\ AND\ NOT(x_1)\ AND\ x_2\ AND\ x_3)$

$OR\ (x_0\ AND\ x_1\ AND\ NOT(x_2)\ AND\ x_3)$

$OR\ (x_0\ AND\ x_1\ AND\ x_2\ AND\ NOT(x_3))$.

f_2 is a minimal Boolean function of 8 variables and is the sum of 58 products (see Appendix for its definition).

$f_3(x_0, x_1, x_2, x_3, x_4, x_5) =$

$(NOT(x_1)\ AND\ NOT(x_2)\ AND\ NOT(x_3)\ AND\ x_4)$

$OR\ (NOT(x_1)\ AND\ NOT(x_2)\ AND\ x_3\ AND\ NOT(x_4))$
$OR\ (NOT(x_1)\ AND\ x_2\ AND\ NOT(x_3)\ AND\ NOT(x_4))$
$OR\ (NOT(x_1)\ AND\ x_2\ AND\ x_3\ AND\ x_4)$
$OR\ (x_1\ AND\ NOT(x_2)\ AND\ NOT(x_3)\ AND\ NOT(x_4))$
$OR\ (x_1\ AND\ NOT(x_2)\ AND\ x_3\ AND\ x_4).EQLOR\ (x_1\ AND\ x_2\ AND\ NOT(x_3)\ AND\ x_4)$
$OR\ (x_1\ AND\ x_2\ AND\ x_3\ AND\ NOT(x_4)),$

where x_i is 0 or 1. f_1 and f_3 are two minimal Boolean functions of 6 variables and the sum of 8 products. Experiments were also done with two other functions f_4 and f_5 which are of the second category functions. f_4 and f_5 are six variables and ten variables functions respectively. For each function, different initial populations were tried, including random (R), column controlled (C) and column and row controlled ($C\&R$). Row controlled populations were not included because the number of symbols in a row (4 to 10 in our experiments) is not large enough to leave enough room for diversity, the probability of generating duplicate rules is very high, and it fails to maximize the number of order 1 schemata. Most of the results were averaged over 8 runs.

Given an input string "$x_0 x_1 x_2 \cdots x_i$", the system responds with either 1 or 0. On-line and off-line performances are used as performance measures [DeJ75]. In our experiments they are defined as

$$P_{on-line} = \frac{(\sum_{t=0}^{T} U_t)}{T},$$

and

$$P_{off-line} = \frac{\sum_{t=0}^{T} (\max_t U_t)}{T},$$

where U_t is the number of the correct responses less the number of incorrect responses at generation t, and T is the current generation.

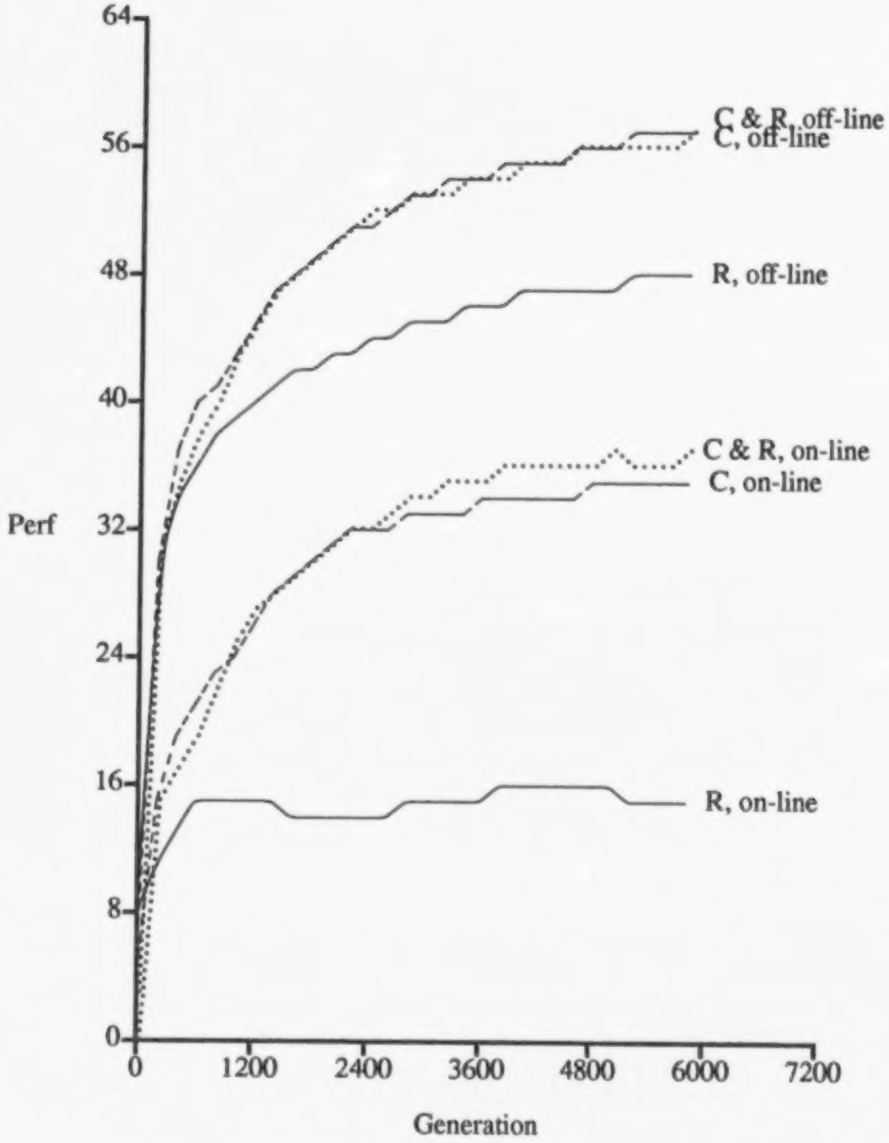

Figure 3.5: f_1 performance, population size 48.

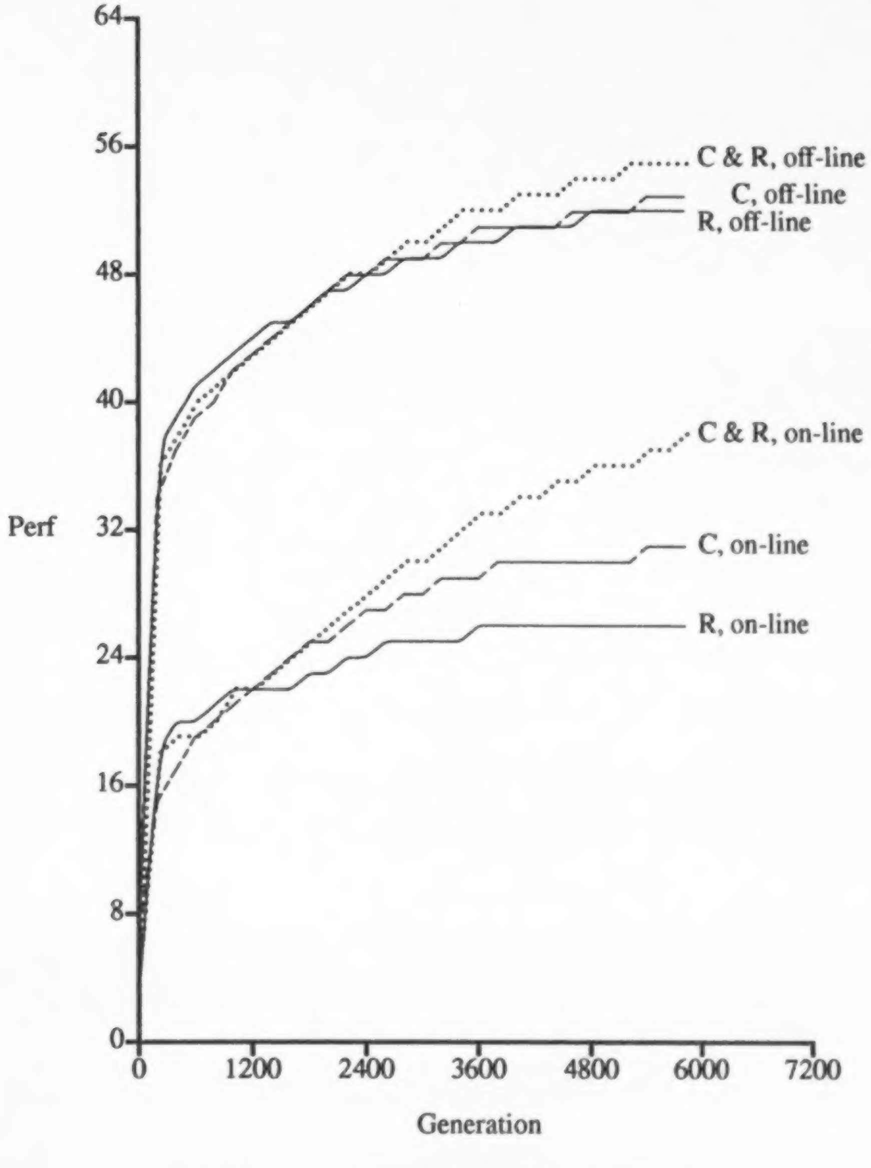

Figure 3.6: f_1 performance, population size 80.

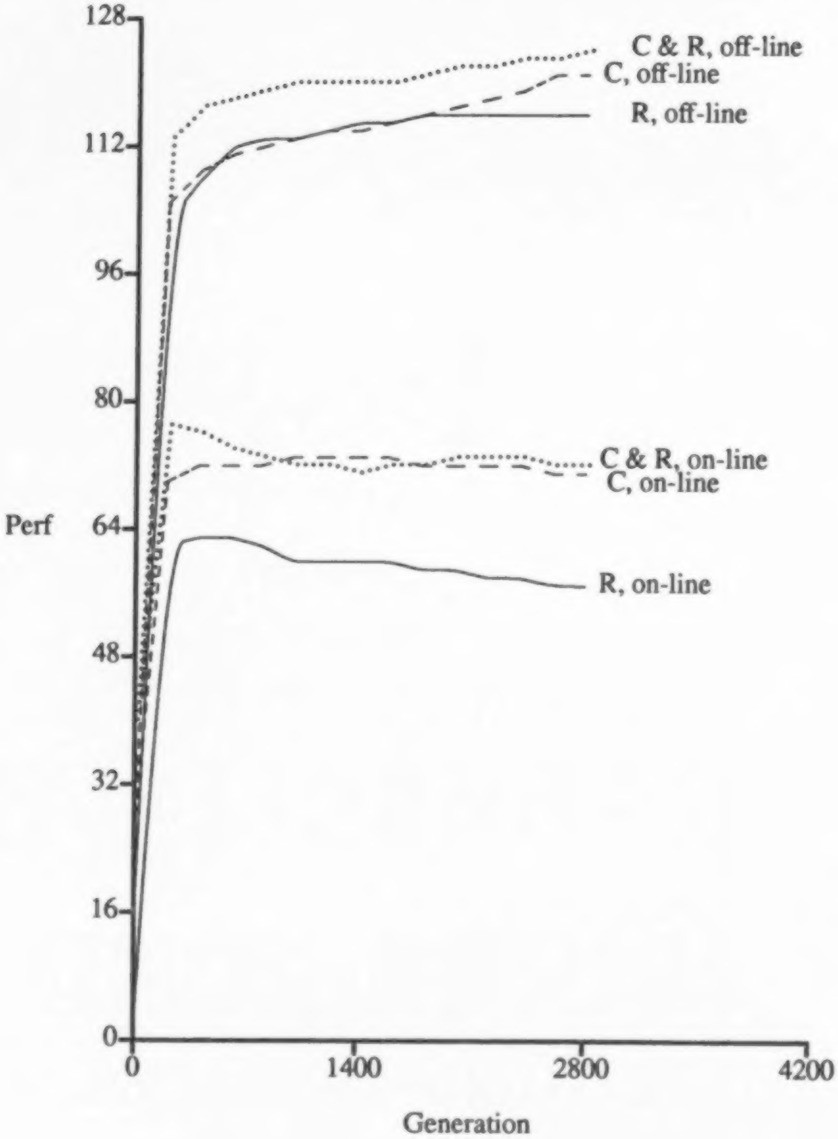

Figure 3.7: f_2 performance, population size 240.

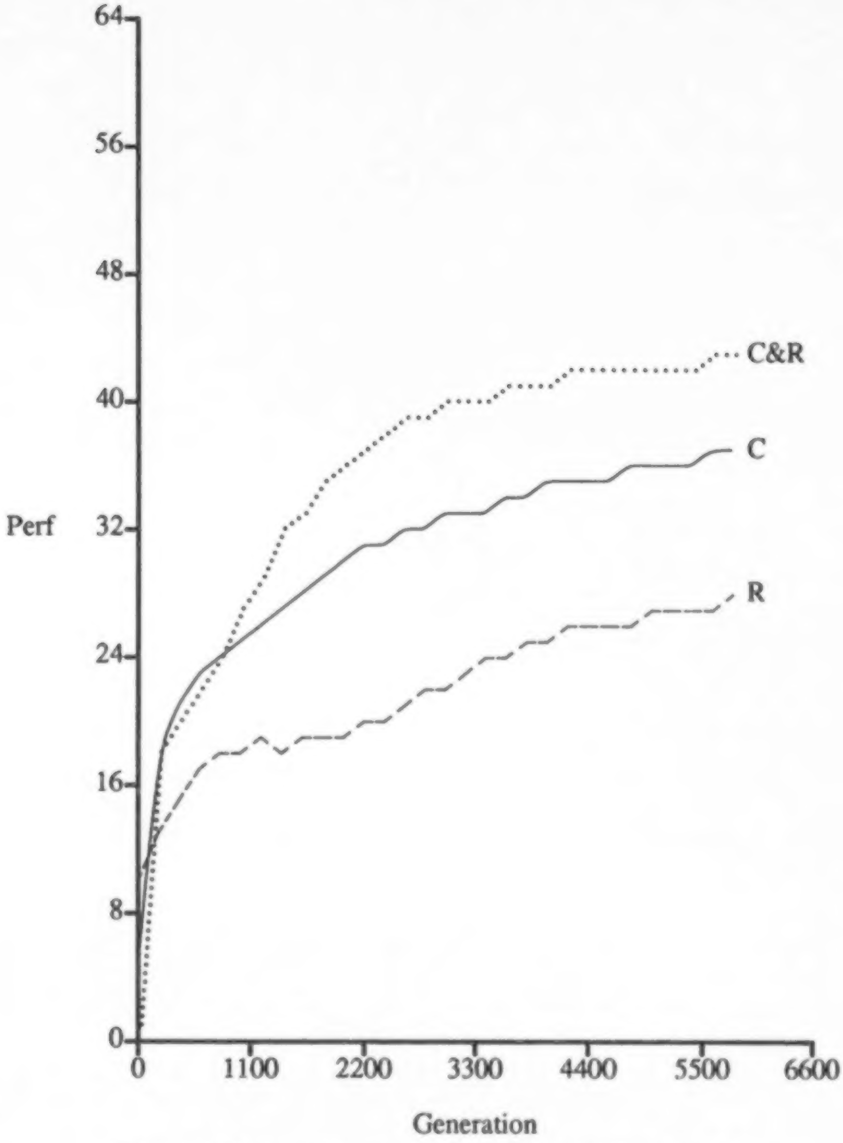

Figure 3.8: f_3 on-line performance, population size 64.

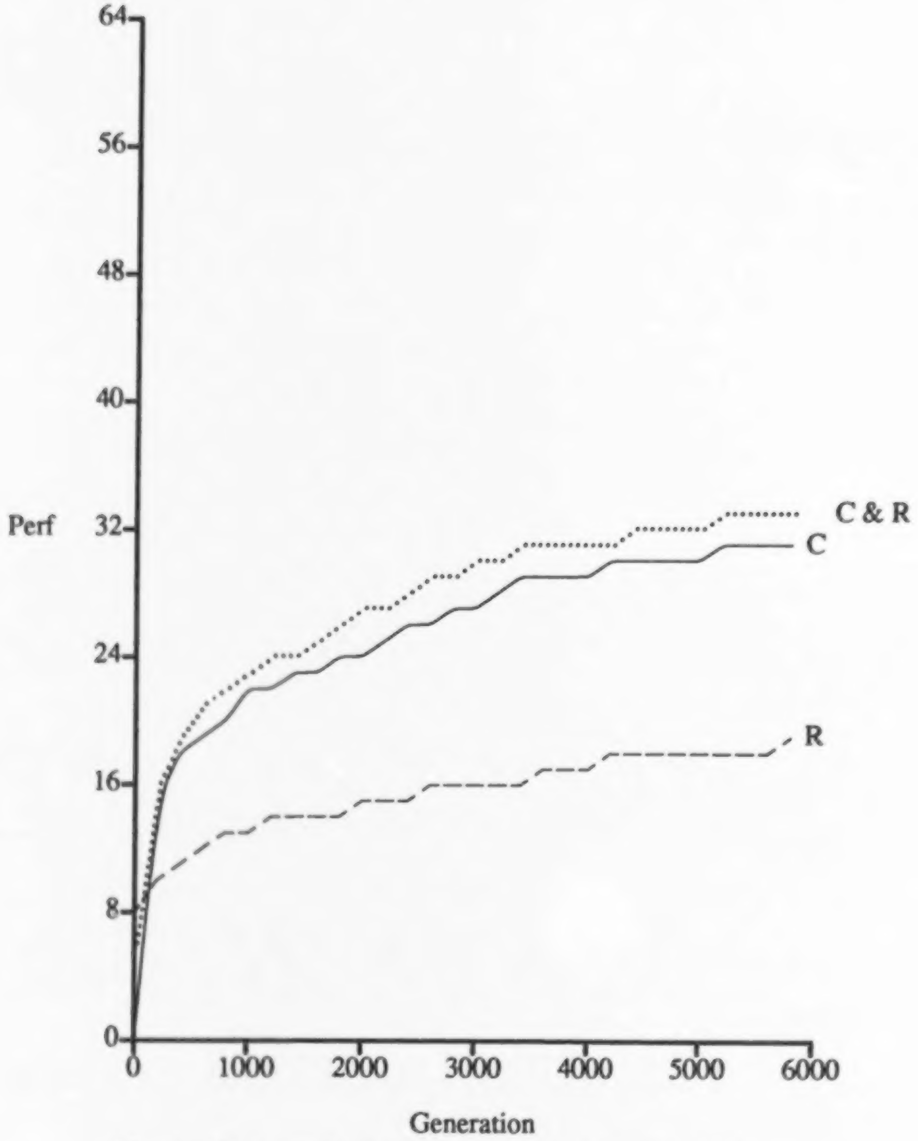

Figure 3.9: f_3 on-line performance, population size 80.

Figure 3.5 shows the performances for f_1 using a population size of 48. The maximum vertical scale value equals the number of matching trials carried out in a generation (this applies to all the figures hereafter). Figure 3.6 shows the results for f_1 when the population size is increased to 80. Figure 3.7 shows the results for f_2. Since this is a more complicated function, a larger population size was used (240) and more trials were performed in each generation (256). Figures 3.8 and 3.9 show the on-line performances of function f_3 with population sizes 64 and 80 respectively. In all the cases, the controlled initial populations reduced the problems of premature convergence and over dominance of sub-solution and significantly outperform the random populations. It also can be seen from Figures 3.5, 3.6 and 3.7 that the differences between the off-line and on-line performances for the random populations are bigger than those for the controlled populations. By the definitions of on-line and off-line performances, the difference between these two kinds of performances is

$$P_{off-line} - P_{on-line} = \frac{\sum_{t=0}^{T}((\underset{t}{max}U_t) - U_t)}{T}.$$

Hence a bigger difference indicates a larger scale of vibration of the performance. So, the bigger differences between the off-line and on-line performances with the random populations may imply that its performance is less stable. The experiments are intended to show the relative performances of different kinds of populations rather than to show the absolute performance.

The experiments also illustrated the following points:

1. With a population that is too small or too big relative to the problem, none of the initial population strategies would perform well. This was shown by the results of all the functions experimented with. For f_1 and f_3, when the population size is greater than or equal to 112, the performances degrade dramatically. So do the

performances of f_2, when the population size is less than or equal to 120. Generally, there is a population size limit for each function, within which, the performances increase when the population size grows. Figure 3.10 shows the changes of the on-line performances along with the changes of the population sizes. In both the random and column & row controlled initial populations, it is shown that the performances decreased when the population size increased to 112.

2. Generally, the column & row controlled initial populations perform better than the column controlled ones. We call the results of random, column controlled and column & row controlled initial populations for an experiment a result set. Each result in a result set is the average of several runs. Among the 24 result sets, only four sets show the reverse, i.e. the column controlled initial populations perform better than the column & row ones. This is consistent with the theories proposed in the previous sections of this chapter. According to the theories, over a large sample of problems, on average, the distance (NS) from the solution set to a column & row controlled initial population will be less than the distance to a column-only controlled population.

3. When the population size is large, a random initial population may perform better than a column controlled one but rarely better than a column & row controlled one. By our theories, including more low-order schemata is the fundamental advantage of a controlled initial population over a random one. But when the population size is large, the probability that a random population contains as many schemata is high. Therefore, the advantage of a controlled initial population diminishes. This point is shown by Figure 3.11.

4. For the second kind of functions, i.e. functions whose solution sets do not contain many various schemata, the results show a different scenario. In this case,

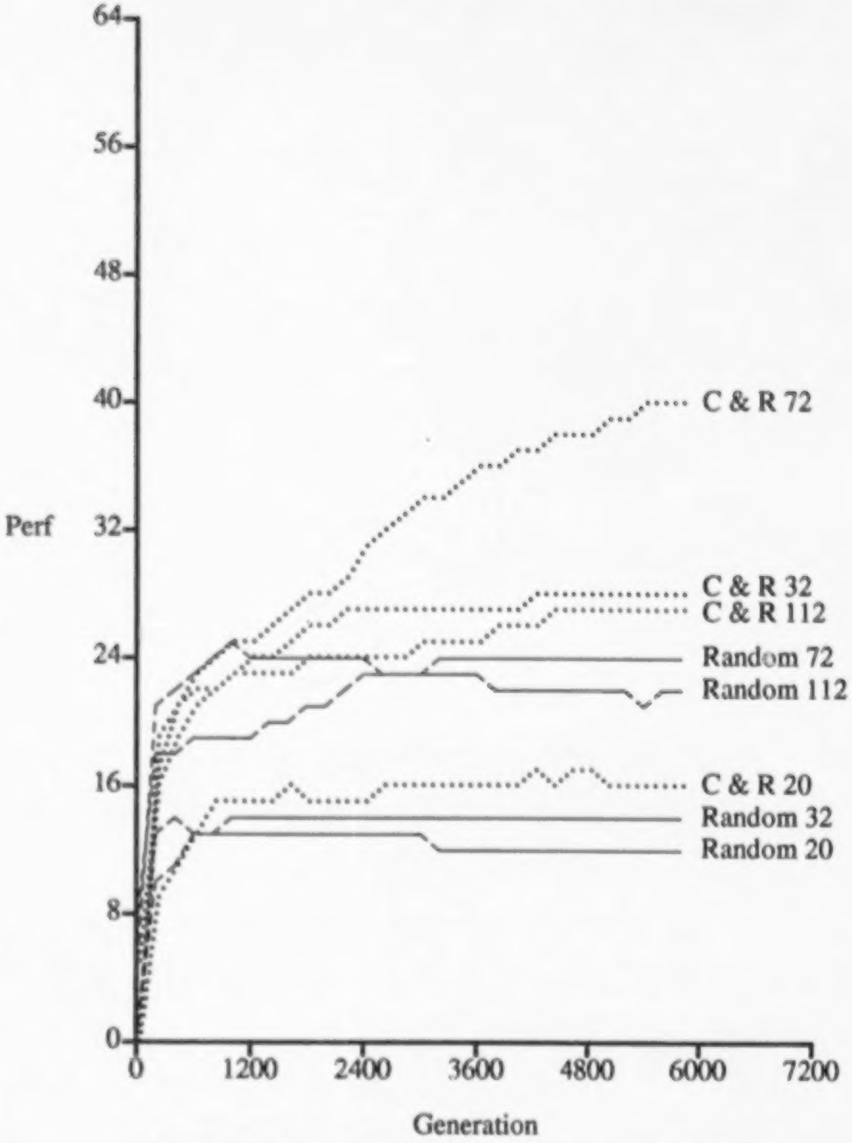

Figure 3.10: f_1 on-line performance by population sizes.

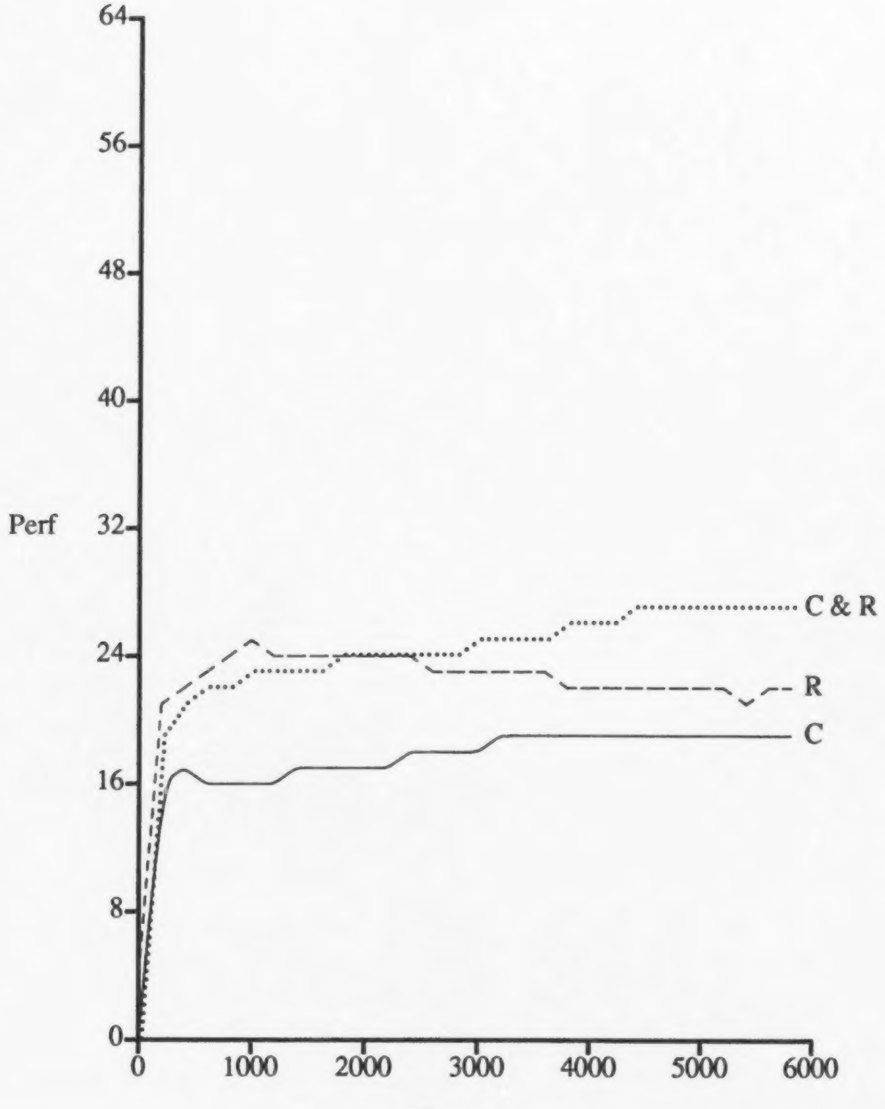

Figure 3.11 f_1 on-line, population size 112.

assuming a solution set would be able to give an answer to every environment state, the majority of the schemata in the solution set must contain many '#'s in order to cover every environment state. Although random initial populations quickly converge prematurely, they still show better performances over the other two initial population strategies. The reason is that, in this case, only a few schemata are needed to cover most of the environment states. As long as the system converged to these schemata, it would respond correctly in most situations. The experimental results of f_5 illustrate this. f_5 is a function of ten variables. Of the 2^{10} instances, only 66 have the value of 1. In the experiments, the random initial populations tended to converge to classifiers such as ##########/0. By the characteristics of the function, even if the entire population contains only ##########/0, the performance would still be around 90%. Thus the performances would always look better unless the controlled ones have achieved over 90% performances.

In all the experiments, the *EDP* s of the controlled initial populations are generally greater than those of the uncontrolled. Since the population sizes are relatively small, the *EDP* s of the controlled initial population are still far less than the maximum values they could have. This accounts for some of the unsuccessful runs with controlled initial populations.

3.4. Conclusions

This chapter has demonstrated the importance of an initial population for the effectiveness and efficiency of a genetic-algorithm-based incremental learning. A general method is proposed for selecting a better initial population. A good initial population helps prevent premature convergence and reduces the problem of over-dominance by one or more sub-solutions. This is accomplished by encouraging the simultaneous emergence of sub-solutions. The basic idea is to include extensive "background knowledge" in the beginning, so that biases to certain classifiers are built up "carefully". Using this method imposes no additional cost on a classifier system and does not introduce any new parameters.

Two measures of the quality of a population, *EDP* and *NS*, are proposed. *EDP* is used to measure the number of order 1 schemata contained in a population. This is important for preventing convergence to wrong solution strings, and guarantees a high degree of parallelism. *NS* is used to measure the distance between a population and a solution(s), which is important for efficiency. These two measures have the potential for being used to control the evolution of a classifier system population. Without any *a priori* heuristic knowledge about a solution, a population that has $\lfloor (l+1)/n \rfloor$ of each symbol per row and m/n of each symbol per column (m being the size of the population, l the length of the string and n the number of symbols used in representing the condition part of the classifiers) is a better initial population than one randomly generated. If some knowledge about the pattern of a solution is available, the rows and the columns of the population array can be arranged using this information to minimize *NS*. The algorithms given for generating initial populations with larger *EDP*s and smaller *NS*s are based on random number generators. Since the test problems do not require large population sizes, *EDP* and *NS* were not used as the performance measures in the experiments.

This initial population theory can be applied when using the *cover operator*. A cover operator is used to generate a classifier whose condition matches the current message. This operation is useful when the population does not respond to the current message. Usually, the number of '#'s contained in the new generated classifier is randomly chosen. This would result in a too general or too specific new classifier. The above theory can be used to decide the number of '#'s in new classifiers.

Chapter 4

Explicit Structural Ties Approach to Classifier Systems

4.1. Introduction

A good initial population (discussed in chapter 3) only provides a good starting "background" for better performances. The dynamics of a system are more decisive on the results. Dynamically, genetic-algorithm-based learning classifier systems suffer from a number of problems that cause poor performance. One of these problems is that the presence of good classifiers and classifier structures in a population are vulnerable to genetic operation disruptions. This difficulty is reflected by the problems of rule clustering and rule association (see chapter 2 for details). The rule clustering problem refers to the phenomenon that all the classifiers in a population may converge to similar patterns, but these patterns do not include all the solutions. The rule association problem refers to the difficulty in achieving cooperation among a set of classifiers. Cooperation includes the building of classifier chains and default hierarchies. As pointed out in chapter 2, these problems have not been solved effectively for complicated tasks.

In this chapter, a method is proposed in which structural ties are used to achieve coherence, impose cooperation and encourage co-adaptation among classifiers. The effects of the coherence and co-adaptation result in parallel developments of sub-solutions, thus reducing the probability of rule clustering. Cooperation among classifiers helps maintain established structures and relations, prevent sub-solutions from interfering with each other and reduce the problem of genetic disruptions. A hierarchically structured classifier system, *HCS*, has been implemented to show the effect of imposing structural ties. At the lowest level, classifiers (*individuals*) are grouped into *families*. If necessary, higher-order structures, such as *communities* of families could be built on top of families. Hereafter, the discussions would be based on a framework of two levels, classifier (individual) level and family level.

In HCS, members of a family cooperate to maximize both the family's strength and the individual member's strength. As in conventional classifier systems, matching is done at the classifier level. However, the bid of a matched classifier is determined not only by the strength of the classifier and its specificity, but also the strength of the family that it belongs to. Therefore, the fates of the family members are bound together. The cooperation among family members are also reflected by restrictions on genetic operations, particularly crossovers among family members. Genetic operations are performed at the family level, meaning that the basic units for a genetic operation are from different families. In HCS, there are two kinds of crossovers. The first one is crossing over two classifiers from different families. This is done by first selecting two families from the population and then selecting one classifier from each family for crossover. The crossover of these two classifiers is the same as that in a conventional classifier system. The second kind of the crossover operation in HCS is crossing over two families. There are two ways of doing this, swapping family members of the two selected families and swapping part of each family member with that of the corresponding member in the other family.

Swapping family members is done through the following steps:

1) Select two families from the population.

2) Randomly choose a number n between 0 and the size of the families (exclusive) as the number of family members to be swapped.

3) Swap the first *n* members of the two families.

For example, assuming the following two three-member families are selected for a crossover operation:

$family_1$: { 00011#/0,
00111#/1,
11####/1 }

$family_2$: { 101111/0,
010000/0,
010100/1 },

and 2 is chosen as the number of members to be swapped, meaning the first two members in the two families are going to be swapped. A crossover between these two families by swapping family members would produce the following two families:

$family_3$: { 101111/0,
010000/0,
11####/1 },

$family_4$: { 00011#/0,
00111#/1,
010100/1 }.

Swapping part of each family member between two families is done through the following steps:

1) Select two families from the population.

2) Randomly select a position between two characters of a classifier. Therefore the classifier is split into two segments.

3) Swap the two segments with the corresponding classifier in the other family.

4) Repeat Steps 1 and 2 on every member of the families.

The following is an example of this kind of crossover. Considering families $family_1$ and $family_2$ given above again. If the position between the third and fourth left characters is chosen, a crossover of this kind would produce

family$_5$: { 10111#/0,
01011#/1,
010###/1 },

family$_6$: { 000111/0,
001000/0,
11#100/1 }.

By considering a family as a matrix of classifiers, the above two kinds of crossover can be viewed as swapping rows of the selected matrices and swapping columns of the matrices respectively.

The experimental results show significant improvements in system performance for *HCS* over conventional classifier systems. The theoretical analysis shows that the family ties in *HCS* enable the system to have greater resistance to the disruptions caused by genetic operations.

The concept of a family superficially appears similar to the idea of *rule sets* in the Pittsburgh method [DeJ88]. The fundamental differences between *HCS* and the Pittsburgh and Michigan methods are discussed in section 4.5.

4.2. Structural Ties Approach

Establishing relationships among a collection of entities usually results in common interests and mutual benefits. Achieving coherence through ties is a universal phenomenon. For example, in business world, merging several small companies into one larger entity often results in a company that is collectively stronger than the sum of the individual members. Other examples include family, group and national ties. Family units make the individual members strongly related by common interests, gains and losses. Therefore their behaviors tend to be for each other instead of against each other. Gaining coherence through establishing structures of relationships is proposed as a method for achieving stability in a classifier system.

Structuring is superimposed on classifier systems by the introduction of hierarchies. Hierarchical Classifier Systems (*HCS*) retain the basic structure of classifier systems, but join several classifiers together as a family. The operands of a crossover can be from the same family or different families. But the probability of a genetic operation between families is much higher than that between classifiers from the same family. The utility of a family is in direct proportion to the sum of the utilities of the classifiers in the family. Matching and firing are still performed at classifier level. However, the bid of a matched classifier is determined not only by the utility of the classifier but also by the utility of the family. Hence, families that contain good classifiers and classifier structures would survive over those containing improper classifiers.

4.2.1. HCS Framework

The syntax of HCS is defined by three-tuple $<E, G, F>$ where:

- $E = \{e_i\}$, where e_i $(i = 1, 2, ..., n)$ is an environment state, which corresponds to the condition part of a classifier. Therefore, n determines the length of a condition string.
- $G = \{g_i\}$, where g_i $(i = 1, 2, \cdots j)$ is a category, which corresponds to the action part of a classifier. Thus j is a factor that determines the length of an action string.
- $F = \{f_i\}$, where f_i $(i = 1, 2, ..., m)$ is a family. f_i consists of a set of *fs* (family size) classifiers.

A classifier contains a condition part and an action part. Both parts are strings over the alphabet {‘0’, ‘1’, ‘#’}.

The computational behavior of *HCS* is defined by the four-tuple <*GA* , *CA* , *SA* , *P*> where:

- *GA* is a set of genetic operations.
- *CA* is a set of credit assignment algorithms.
- *SA* is a set of selection algorithms used for choosing classifiers for firing, families for deletion, and families or classifiers for the genetic operations.
- *P* is a set of system-dependent parameters which includes the crossover rate, population size (*ps*), the number of classifiers contained in a family (*fs*), etc.

In *HCS*, classifiers are bound into families. Mainly, families, instead of classifiers, are the objects of genetic operations. Within a family, classifiers are related to each other by the common fate imposed on them. Naturally, the competitions among these classifiers are restricted, and the competitions inside a family are different from those among the families. These characteristics of *HCS* make it possible to reduce the effect of the rule clustering and rule association problems.

4.3. Reducing the Rule Clustering and Rule Association Problems

The main causes of the rule clustering problem are the competitions among useful patterns or schemata, and the disruption of useful patterns by genetic operators. For example, assume classifiers "###1/0" and "0001/1" are members of the solution set. In conventional classifier systems, crossover between these two classifiers may produce children "##01/0" and "00#1/1". But "00#1/1" implies 0011/1 and therefore is not compatible with the solution set. However, in *HCS*, if the two classifiers stay in the same family, genetic operations and competitions between the two classifiers would be restricted, reducing the likelihood of producing incompatible classifiers. The following is another example of the competitions among useful schemata. Suppose classifiers

"00##/1", "10##/1", "11##/1" and "01##/0" are the sub-solutions. Then there is a tendency that the instances of schema ****/0 would become fewer and fewer in the population, and the instances of ****/1 would become more and more. Eventually, schema ****/0 may disappear from the population. However, in *HCS*, instances of schema ****/0 could be in the same family with instances of ****/1, therefore, co-exist with the instances of ****/0.

Premature convergence is another reason for the clustering problem. Let l be the length of the classifier strings, s the size of the solution space, S the size of the search space and ps the population size. The probability that at least one solution classifier c is included in the initial population is $p_c = 1 - (\frac{S - s}{S})^{ps}$. The search space in a traditional classifier system over the alphabet {'0', '1', '#'} is 3^{ℓ}. Therefore, $p_c = 1 - (\frac{3^l - s}{3^l})^{ps}$. If other useful classifiers appear much later, c would become a super-classifier with a much higher utility in the population. Consequently, the population tends to converge to this string prematurely. In *HCS*, assuming fs is the number of the classifiers contained in a family, the search space is $S \geq \binom{3^l}{fs}$. The probability that at least a correct family is contained in an initial population is much smaller than p_c in a conventional classifier system when $fs > 1$. Hence, the chance that one family dominates the population is smaller. The point here is not to claim that larger search space is better. Rather, we are trying to point out that it is easier to encourage co-adaptation in *HCS*. This point can be seen from a different view. Suppose the probability that a classifier is a good classifier is p, then the probability that a family is a good family is p^{fs} assuming that every classifier in a good family is a good classifier. So, it is harder for a family to be dominant than for a classifier.

HCS also has the potential of reducing the rule association problem. In *HCS*, it is possible to group classifiers into families by a relation r_i. That is, use r_i as a constraint when forming families. By selecting proper relations r_i, classifiers can be grouped into families that contain reasoning chains or default hierarchies. Thus these associated classifiers are bound together explicitly. Therefore, their relationships can be considered locally inside the family. For example, consider a classifier system with a bidding scheme that favors the exception classifiers. With a default hierarchy of "###1/0" and "0001/1", the system would also favor classifiers such as "0011/0" and "0101/0", which are covered by the default classifier "###1/0". In *HCS*, if the bidding scheme favors the exception classifier only locally in the default hierarchy family {"###1/0", "0001/1"}, the ***starving general classifiers*** phenomenon (see chapter 2) would not occur as a result of giving higher priority to the exception classifiers. It is interesting to note that when classifier chain is used as the relation to form families, *HCS* is similar to the idea of corporate classifier system proposed in [WiG89] when the concern of the generation and maintenance of long classifier chains was addressed. However, the corporate classifier system emphasizes structures existing in a task wherevers *HCS* emphasizes imposing structures onto a task. In *HCS*, a chain, especially a long chain, would benefit from the imposed structure even when the members of the chain do not reside in the same family (for more discussion, see the last section of this chapter).

4.4. Experiments with HCS

4.4.1. Implementation of the HCS Framework

HCS has been implemented as follows:

- $E = e_i$, where e_i ($i = 1, 2, ..., 2^l$) is the binary string representation of a number between 0 and 2^l, l is the length of the strings.

- Category $G = \{0, 1\}$.
- $F = f_i$, where f_i $(i = 1, 2, ..., f)$ is a family that contains fs classifiers.

Classifiers are grouped into families randomly. The genetic operations used are crossover and the covering operator. The covering operator replaces the condition part of a classifier chosen from the population with the message on the message list. This operator is used with a probability when no classifier in the population matches the message. The covering operation is a form of mutation. The crossover operation in HCS is different from the one in conventional classifier systems. The basic credit assignment algorithm is used, i.e. when a classifier responded correctly, some credit is given to the classifier. When a classifier responded incorrectly, some credit is taken from the classifier. Two levels of credit measurements are used; the utility of a classifier and the utility of a family. The utility of a family is the sum of the utilities of all the classifiers contained in the family. When any of the classifiers in a family gains, the family gains. Whether a matched classifier would be fired is determined by its bid, which is calculated by the following formula:

$$bid = \frac{k \times u_c \times u_f \times sp}{2^{l-sp}},$$

where k is a constant, u_c is the utility of the classifier, u_f is the utility of the family containing the classifier and sp (specificity) is the number of the 'non-#'s in the condition part of the classifier.

With a small possibility of operating at the classifier level (0.001 is used in our experiments), i.e. two classifiers are selected directly from the population instead of from two selected families. In most cases, the operands of crossover operations are families, i.e. two families are selected first for a crossover operation and the classifiers in a family are restricted from crossing over with each other. The so called *roulette* *wheel selection* [Gol89a] is used to select candidates for crossover operations. Families (or classifiers in the case of crossing over at the classifier level) with higher utilities have a greater chance of being selected. In each generation, families with the lowest utilities are selected as candidates for deletion.

Since the test problems we use (see section 4.4.2) do not require building up classifier chains, no internal message is used. At the end of each match-activation cycle, a new environmental message is generated and placed onto the message list. The match-activation is done in the following way:

1. The message on the message list matches against the condition part of each classifier.
2. Conflict resolution is done among all the matched classifiers. The classifier with highest bid is chosen for activation.
3. The action part of the activated classifier is used to check against with the environmental feedback. If the action matches the feedback, the classifier earns positive payment from the environment, otherwise the classifier earns negative payment.
4. The strength of the activated classifier at the end of cycle $t + 1$, S_{t+1}, is determined by the formula: $S_{t+1} = S_t - Bid + Payment$, where Bid is the bid of this classifier in this cycle and $Payment$ is the environmental payment in this cycle. The strength of the family that contains the activated classifier is calculated by the formula: $Sf_{t+1} = Sf_t - Bid + Payment$, where Sf_t is the strength of the family at the end of cycle t, Bid and $Payment$ are the same as above.

4.4.2. Test Problems and Performance Measures

HCS was tested by having the system learn Boolean functions. This domain was chosen for the following reasons:

1) The problem can be precisely defined.

2) It is easy to represent the problem in a classifier system.

3) The environment states can be controlled easily. The number of the states can be determined by the number of variables in the function. The pattern of the environment feedback can also be controlled.

4) The extent of environmental noise is controllable. It can range from no noise to very noisy.

5) The results of a learning is a set of classifiers with various schemata at the same positions in the classifier strings. The degree of schemata variety and the number of classifiers needed to form a solution set can be changed by selecting different functions.

For example, for function

$$f_a(x_0, x_1, x_2, x_3) = x_0 \; AND \; x_1 \; AND \; x_2 \; AND \; x_3 \, ,$$

the sub-solutions are

{1111/1, ####/0},

assuming that the default hierarchy mechanism works well. For function

$$\begin{aligned} f_b(x_0, x_1, x_2, x_3) = {} & (x_0 \; AND \; x_1) \; OR \; (x_2 \; AND \; x_3) \; OR \\ & (NOT(x_0) \; AND \; NOT(x_1)) \; AND \; (NOT(x_2) \; AND \; NOT(x_3)), \end{aligned}$$

the sub-solutions are

{11##/1, ##11/1, 00##/1, ##00/1, ####/0}.

These two examples illustrate how easy it is to vary the size of a solution set and the number of schemata contained in a solution set. Our goal is to improve the performance and stability of tasks whose solution set contains various schemata at the same positions in the classifier strings.

Many functions were used in our experiments, including the following five functions used in this chapter for illustrative purposes:

$$\begin{aligned}
f_1(x_0, x_1, x_2, x_3) = {} & (x_0 \mathit{AND}\ x_1) \mathit{OR}\ (\mathit{NOT}(x_0) \mathit{AND}\ \mathit{NOT}(x_1)) \mathit{OR} \\
& (x_2 \mathit{AND}\ x_3) \mathit{OR}\ (\mathit{NOT}(x_2) \mathit{AND}\ \mathit{NOT}(x_3)). \\
f_2(x_0, x_1, x_2, x_3, x_4) = {} & (x_0 \mathit{AND}\ x_1) \mathit{OR}\ (x_2 \mathit{AND}\ x_3 \mathit{AND}\ x_4). \\
f_3(x_0, x_1, x_2, x_3, x_4, x_5) = {} & (\mathit{NOT}(x_1) \mathit{AND}\ \mathit{NOT}(x_2) \mathit{AND}\ \mathit{NOT}(x_3) \mathit{AND}\ x_4) \mathit{OR} \\
& (\mathit{NOT}(x_1) \mathit{AND}\ \mathit{NOT}(x_2) \mathit{AND}\ x_3 \mathit{AND}\ \mathit{NOT}(x_4)) \mathit{OR} \\
& (\mathit{NOT}(x_1) \mathit{AND}\ x_2 \mathit{AND}\ \mathit{NOT}(x_3) \mathit{AND}\ \mathit{NOT}(x_4)) \mathit{OR} \\
& (\mathit{NOT}(x_1) \mathit{AND}\ x_2 \mathit{AND}\ x_3 \mathit{AND}\ x_4) \mathit{OR} \\
& (x_1 \mathit{AND}\ \mathit{NOT}(x_2) \mathit{AND}\ \mathit{NOT}(x_3) \mathit{AND}\ \mathit{NOT}(x_4)) \mathit{OR} \\
& (x_1 \mathit{AND}\ \mathit{NOT}(x_2) \mathit{AND}\ x_3 \mathit{AND}\ x_4) \mathit{OR} \\
& (x_1 \mathit{AND}\ x_2 \mathit{AND}\ \mathit{NOT}(x_3) \mathit{AND}\ x_4) \mathit{OR} \\
& (x_1 \mathit{AND}\ x_2 \mathit{AND}\ x_3 \mathit{AND}\ \mathit{NOT}(x_4)). \\
f_4(x_0, x_1, x_2, x_3, x_4, x_5) = {} & (x_0 \mathit{AND}\ x_1 \mathit{AND}\ x_2) \mathit{OR}\ (x_3 \mathit{AND}\ x_4 \mathit{AND}\ x_5) \mathit{OR} \\
& (x_1 \mathit{AND}\ x_2 \mathit{AND}\ x_4) \mathit{OR}\ (x_0 \mathit{AND}\ x_3 \mathit{AND}\ x_5) \mathit{OR} \\
& (x_0 \mathit{AND}\ x_2 \mathit{AND}\ x_4) \mathit{OR}\ (x_0 \mathit{AND}\ x_1 \mathit{AND}\ x_4) \mathit{OR} \\
& (x_1 \mathit{AND}\ x_2 \mathit{AND}\ x_3) \mathit{OR}\ (x_2 \mathit{AND}\ x_4 \mathit{AND}\ x_5) \mathit{OR} \\
& (x_2 \mathit{AND}\ x_3 \mathit{AND}\ x_5) \mathit{OR}\ (x_2 \mathit{AND}\ x_3 \mathit{AND}\ x_4) \mathit{OR} \\
& (x_0 \mathit{AND}\ x_4 \mathit{AND}\ x_5) \mathit{OR}\ (x_0 \mathit{AND}\ x_3 \mathit{AND}\ x_4) \mathit{OR} \\
& (x_0 \mathit{AND}\ x_1 \mathit{AND}\ x_5).
\end{aligned}$$

f_5 is the same as the f_2 in chapter 3, which is a minimal Boolean function of 8 variables and is the sum of 58 products (see Appendix 1 for the definition).

To see a difficulty of conventional classifier systems, consider the solution set for f_1. In the best situation, the solution set would contain the following five classifiers:

{00##/1, 11##/1, ##00/1, ##11/1, ####/0}.

Assume that the default hierarchies work well; the exceptions can always be fired when the input matches both the default classifiers and the exception classifiers. If the first and second classifiers are selected for crossover and the crossover points are between the first and second bits, the crossover will produce 01##/1 and 10##/1, which are not compatible with the existing solution set. Since there is no effective way to guarantee a correct credit assignment to the new classifiers generated by a genetic operation, there is no guarantee that the disruptive classifiers would not be strong enough to compete with other classifiers. Hence, when the disruptive ones survive over the good ones, performance would be adversely affected and the system is unstable.

The experiments were done with two kinds of performance measures. The first kind is the ever-used online performance which is the same as that defined in chapter 3, i.e.

$$P_{online} = \frac{\sum_{t=1}^{T}(C_t - I_t)}{T},$$

where C_t is the number of the correct responses at generation t, I_t is the number of the incorrect responses at generation t and T is the current generation. In order to describe the instantaneous performance at each generation, the second measure is defined as

$$P_{ins} = C_t.$$

This measure can be used to evaluate the stability of a system in terms of the differences between generations.

4.4.3. Results and Analyses

The performance of *HCS* has been examined with different family and population sizes. The family size refers to the number of classifiers contained in a family. The population size is the number of families multiplied by the family size. According to the size of the search space of a problem, in each generation, 64 and 128 trials are carried out for f_1 and f_2 respectively. 256 trials are conducted in each generation for f_3, f_4 and f_5. As described previously, several kinds of crossover can be performed in HCS. The experiments are first done with crossover on classifiers. In single-member family case, this is the same as that in conventional classifier systems. In multiple-member family cases, this is done in the following steps:

(1) Select two families according to their strengths, the bigger the strength, the higher the possibility of being selected.

(2) Select one classifier from each of the selected families. This is done randomly in the experiments.

(3) Carry out the conventional crossover.

Figures 4.1, 4.2 and 4.3 show representative performance results for learning three of the functions given in the previous section, with family sizes varying from 1 to 4. Each performance line shows the average of five runs generated with different random number seeds. The horizontal axis measures the number of generations elapsed. The vertical scale in the figures represents the performance.

All three figures show consistent improvements with *HCS* (*fs* = 2, 3, 4) compared with traditional classifier systems (*fs* = 1). These cases are representative of most of our experimental runs. In Figures 4.1 and 4.2, the performances of family sizes 2, 3 and 4 are not significantly different. The major benefits are obtained using families of size 2, in part because the population sizes are too big relative to the complexity of the problem being solved. This point is addressed later in this section.

The performance for learning f_5 shown in Figure 4.3 is not as good as those for learning f_2 and f_3 shown in Figures 4.1 and 4.2 respectively. The reason for this is

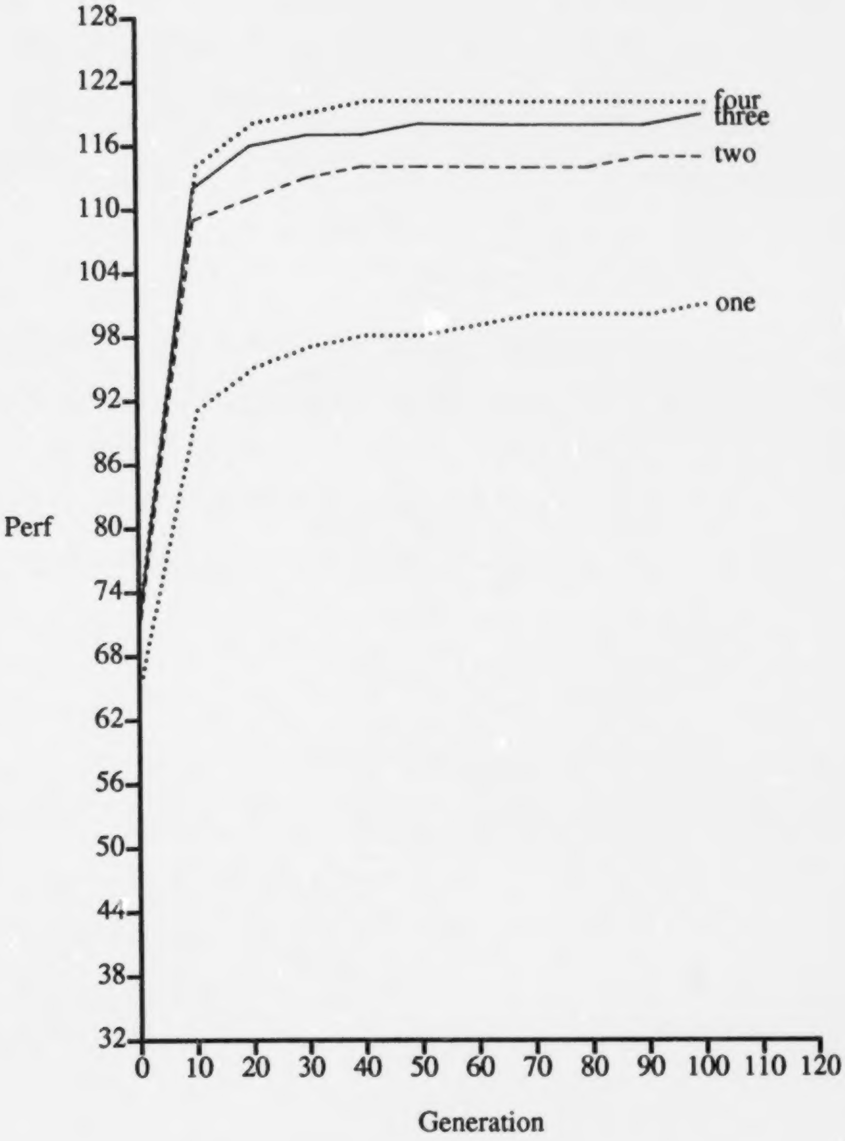

Figure 4.1: f_2 on-line performance, ps = 240.

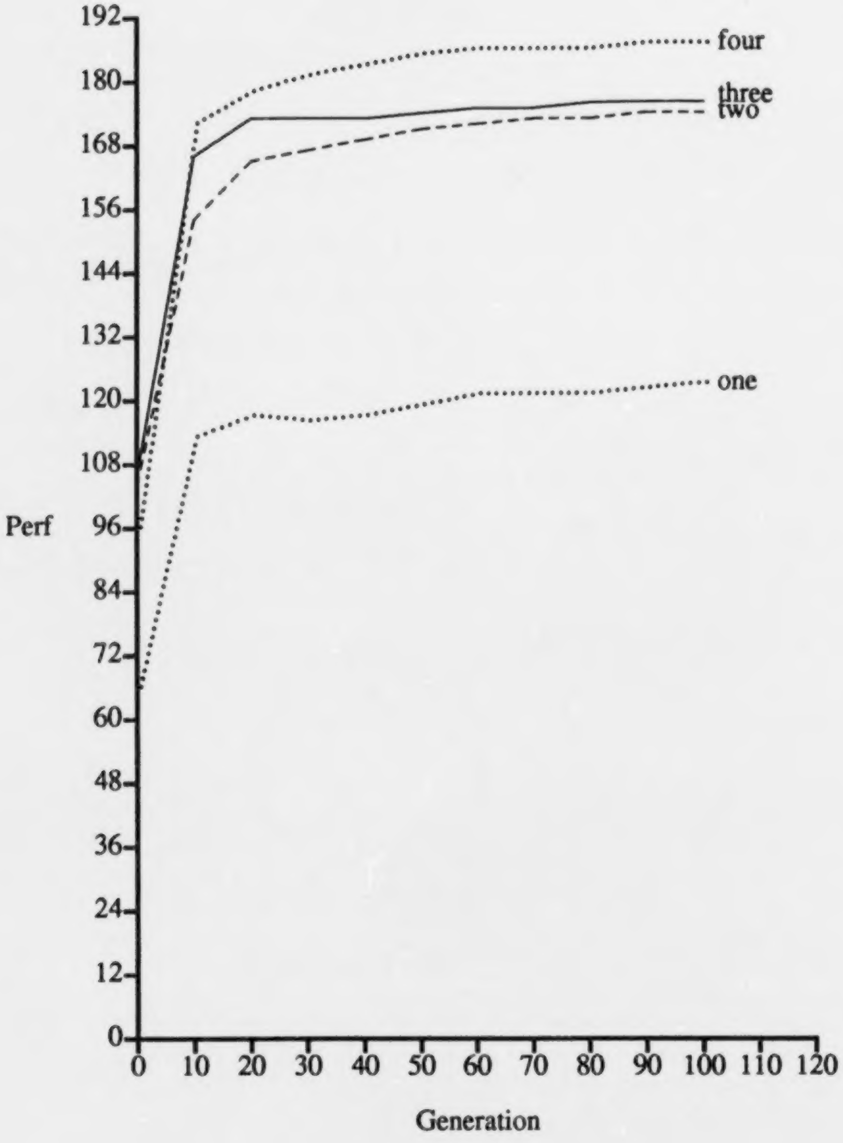

Figure 4.2: f_3 on-line performance, ps = 360.

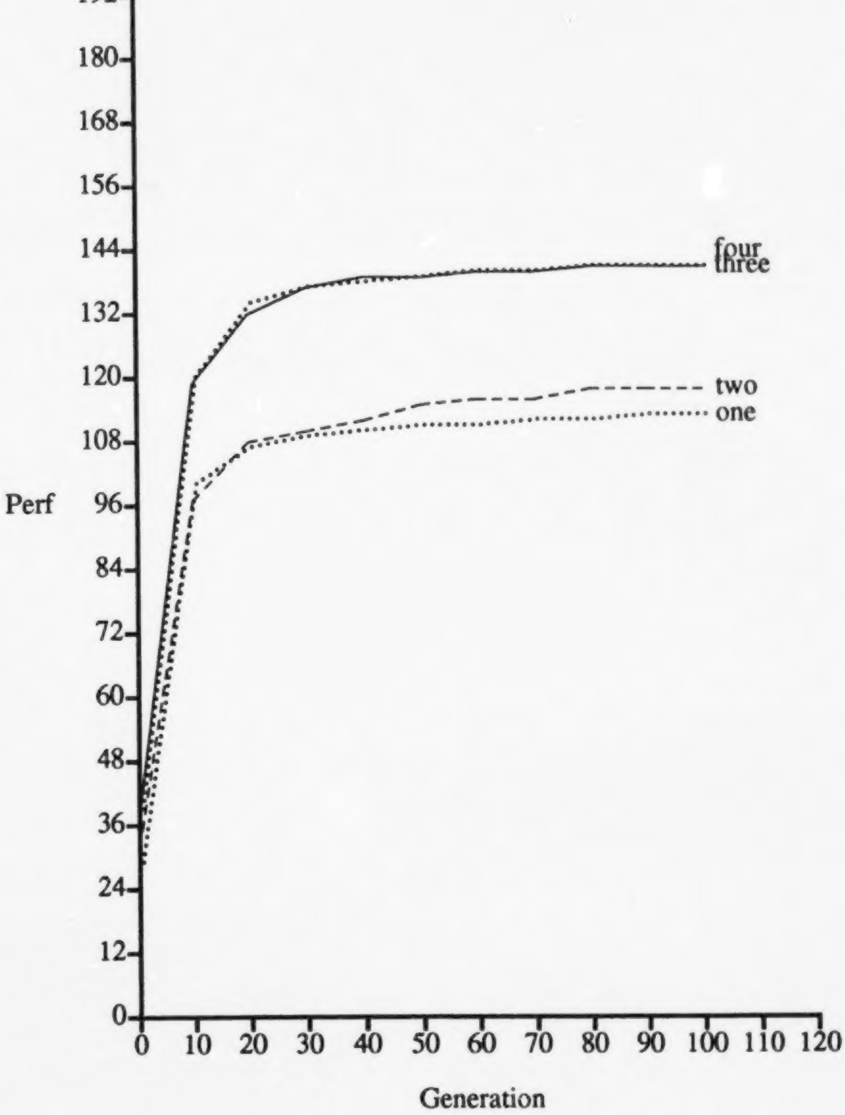

Figure 4.3: f_5 on-line performance, ps = 720.

that f_5 is a much more complicated Boolean function than f_2 and f_3 are. A population of 720 is too small for f_5 given the complexity of the solution set. This point shows that the size of a solution set (complexity of the problem being solved) and the size of a population are important factors in determining performances. However, in all the three figures, Figures 4.1, 4.2 and 4.3, performances relative to each family size clearly demonstrate the significant advantages gained by using families.

The experiments show that within a range (determined experimentally per problem), the larger the population size, the better the performance of *HCS*. Classifier systems ($fs = 1$) generally show the same trend as well, but appear to be not as stable as *HCS*. Figures 4.4 and 4.5 illustrate the performance of families as a function of population size ($ps = 120, 240, 360, 720, 1440$). An example of classifier system instability is shown in Figure 4.4 where the performance of the classifier system does not necessarily increase as the population size grows. For example, performance for $ps = 360$ is *less* than that for populations of size 120 and 240.

An interesting point to note is that often a smaller population of families can out-perform a classifier system with more classifiers. For example, in Figure 4.4, a population of 120 families ($ps = 240$) out-performs classifier systems with up to 720 classifiers. This example is not an isolated experience.

Larger populations help reduce the premature convergence problem in the early generations of a run. Classifier rules representing different sub-solutions can be established in parallel, accounting for the better performance of larger populations. However, for classifier systems, in the later generations of a run, these co-existing sub-solutions may disrupt each other. Repeating the example given earlier, if a solution set for f_1 is

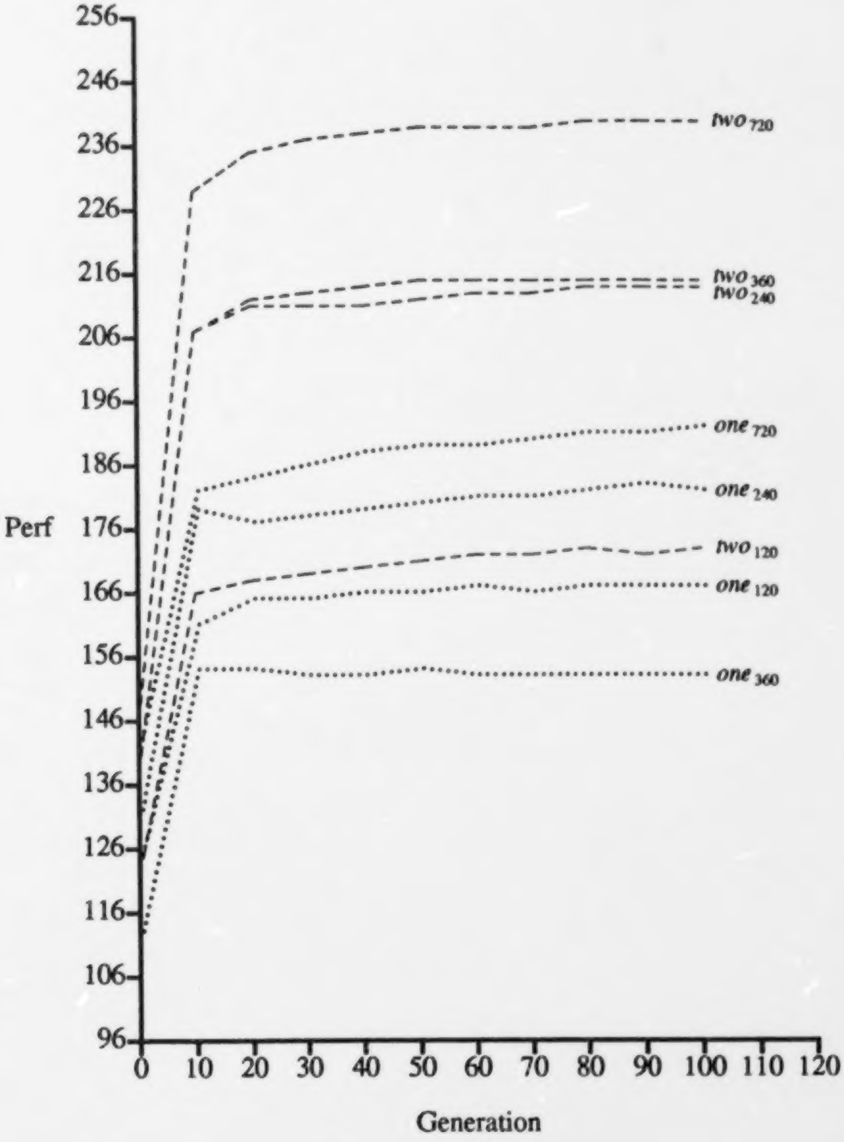

Figure 4.4: f_4 on-line performance.

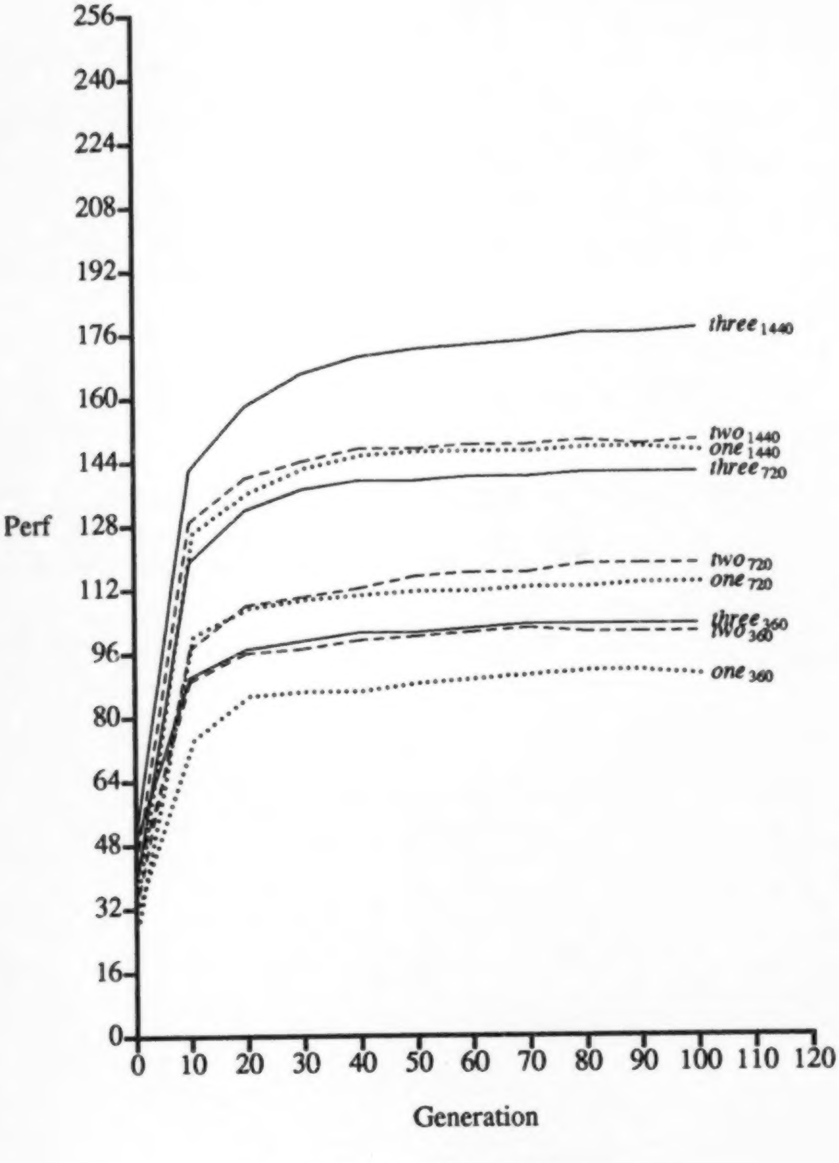

Figure 4.5: f_5 on-line performance.

{ 00##/1, 11##/1, ##00/1, ##11/1, ####/0 },

then using the first two rules and crossing over, between the first and second bits, will result in new, undesirable rules 01##/1 and 10##/1. In *HCS*, if sub-solutions are contained in a family, then family ties make the probability of crossover within those sub-solutions small.

Consider two populations, one of which is relatively small and the other relatively large in comparison to the population required to solve the problem effectively. Comparisons between Figures 4.2 and 4.6 show that in the smaller population, the differences between the performances of larger family sizes (*fs* = 4, for example) and smaller family sizes (*fs* = 2, for example) are more significant. There are two major reasons for this. First, when the population size is larger, as explained previously, there is more room for sub-solutions to evolve, negating the benefits of having large families. Second, when the population size is too small, the problems of premature convergence and genetic disruption are more severe. In this case, larger family sizes are more effective in reducing these problems. The implication of this is that, when the sizes of populations are restricted in practice, larger family sizes can be used to achieve higher performance for problems that require a big population to be solved effectively in a conventional classifier system.

Figure 4.7 illustrates that if the family size is too large relative to the population size, the number of resources available in the population is limited and performance degrades. In Figure 4.7, with a *ps* of 16, the performances decrease when *fs* = 5 and 6. Another reason for the degradation is related to the parasite problem. This problem refers to the phenomenon that a classifier with a low utility is able to offer a high bid because of the high utility of the family that the classifier belongs to. This may happen in a multiple-member family system for two reasons. First, the bid of a classifier is in

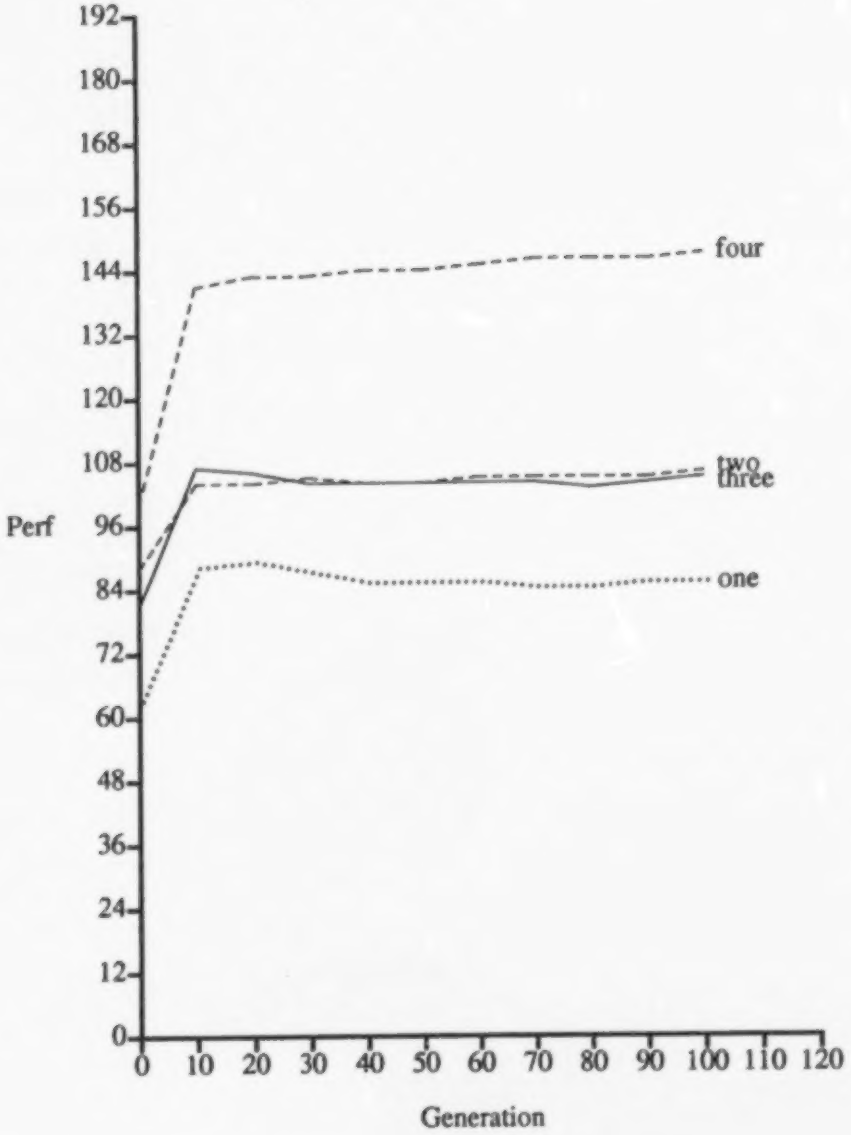

Figure 4.6: f_3 on-line performance, ps = 120.

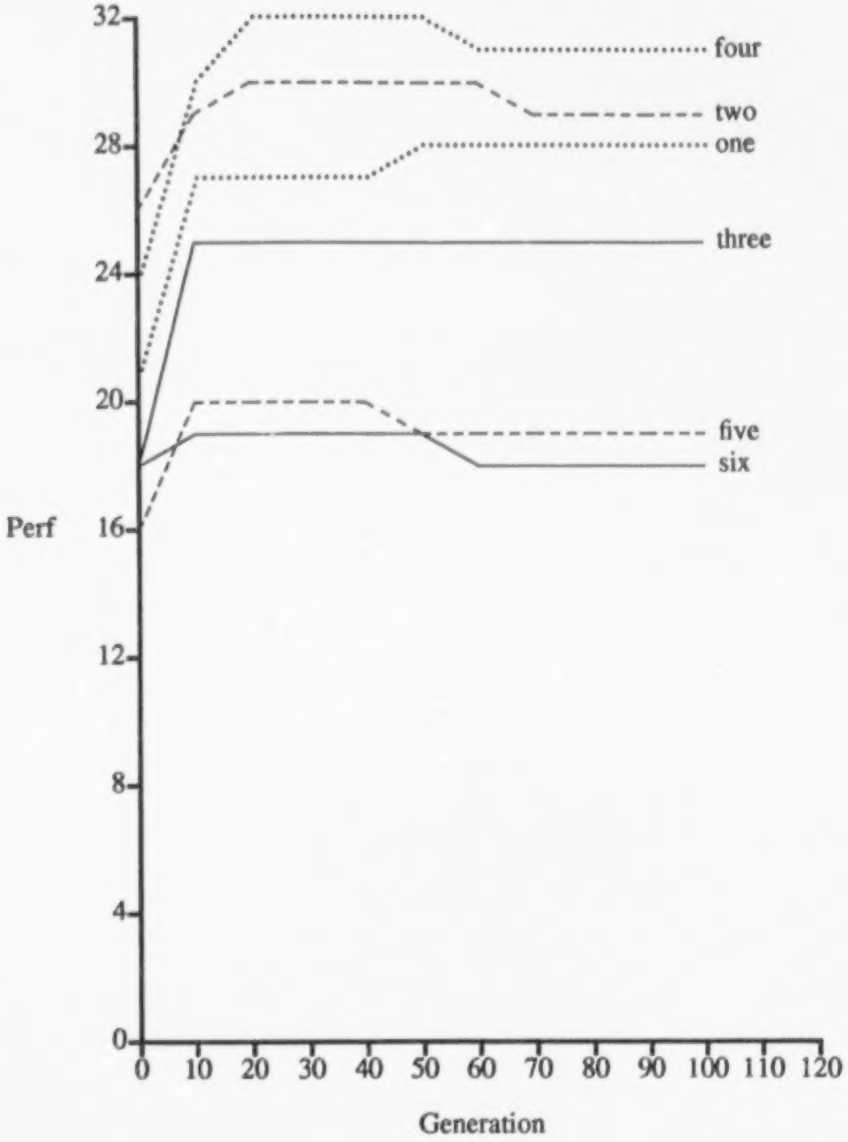

Figure 4.7: f_1 on-line performance, ps = 16.

direct proportion to the utility of the family it resides in. Second, the objects for deletion are families instead of classifiers. Therefore, if the majority of the family members have high utilities, some weak members would survive over the competitions with the members of other families. This problem is more severe when the family size is large since each family may have more than one parasite and each parasite can take advantage of more than one family member.

As is obvious in Figure 4.7, the performances of the runs with multiple-member families are as unstable as, or worse than, those of the runs with single-member families. This phenomenon relates to the influence of the population size. It has been suggested that a population size that is too small could cause poor performance [Gol89b, SCE89, ZhG89]. Note that the population size in the figures presented here are the number of classifiers in the population. This has different meanings when used in single-member and multiple-member family systems. In the single-member family case, a classifier is the basic structure of the system, hence the population size reflects the number of structures as a resource available to the population. In multiple-member family systems, the population is made up of families. Genetic operations are conducted between families and the deletion action is done over families instead of classifiers. Now the population size does not reflect the real number of structures as a resource available to the population. The actual number of structures in the population is $\frac{ps}{fs}$. In Figure 4.7, the population size is 16. Therefore, for 2-member families, the number of structures is $\frac{16}{2} = 8$, and for 3-member family runs, it is $\frac{16}{3} \approx 5$. For 4-, 5- and 6-member family runs, the numbers of structures are 4, 3 and 2 respectively. The small numbers of structures in the populations account for the unstable performances.

The necessity for large family sizes is problem dependent. Generally, bigger

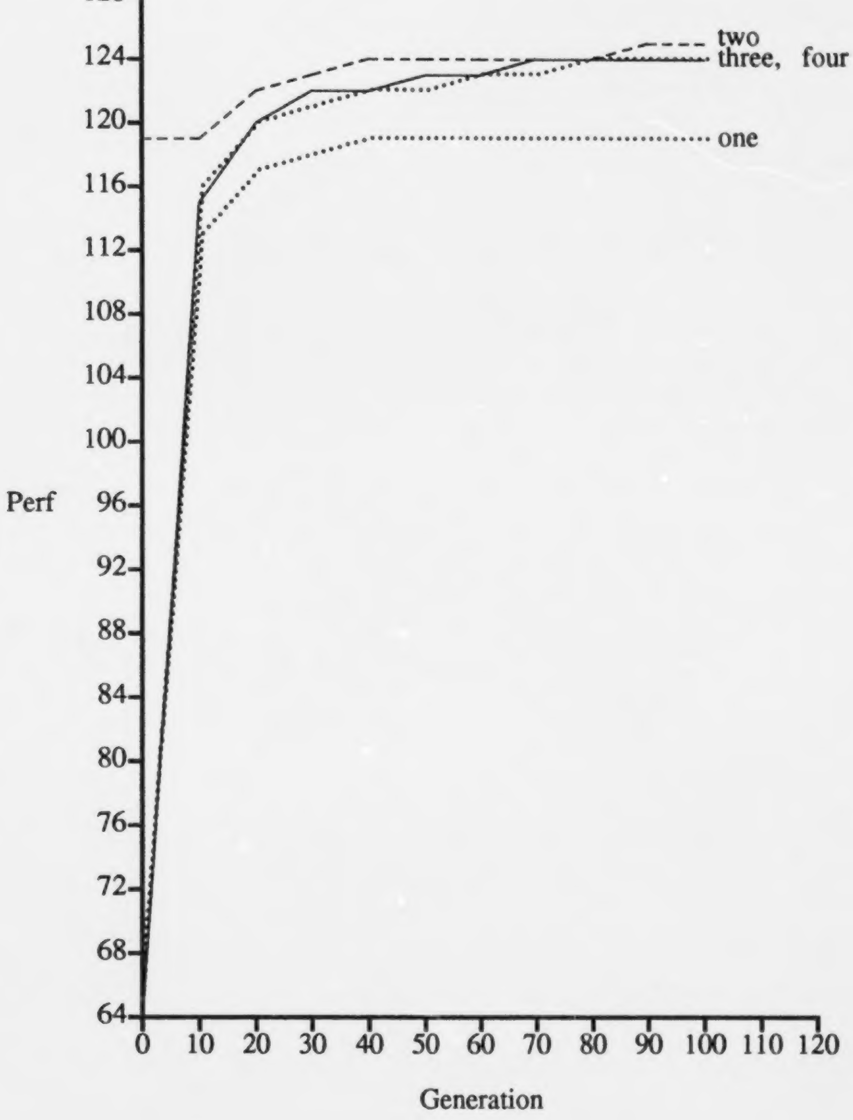

Figure 4.8: f_2 on-line performance, ps = 720.

problems (in this case, functions of more variables and more products) need larger family sizes. However, larger families may not bring any benefits to smaller problems. Figure 4.8, in which the performances for $fs = 2$, 3 and 4 are roughly identical, shows this. In this case, larger families would increase the search space and therefore probably slow down the learning process. As pointed out previously, the search space is greater than $\begin{bmatrix} 3^l \\ fs \end{bmatrix}$. Thus when $l = 4$, the search space for a system with $fs = 2$ is greater than $\begin{bmatrix} 3^l \\ 2 \end{bmatrix} = \frac{81 \times 80}{2} = 3240$, and for a system with $fs = 3$, the search space is greater than $\begin{bmatrix} 3^l \\ 3 \end{bmatrix} = \frac{81 \times 80 \times 79}{2 \times 3} = 85320$. It is obvious that the search space increases significantly when the family size increases only by 1. The problem of choosing an appropriate family size, given an arbitrary problem, is an interesting future research topic.

Figures 4.9 and 4.10 show two typical results described by the instantaneous performance (P_{ins}), the correct responses of a generation. Each result is the average of five runs. The improvements of multiple-member families over single-member families are not as significant as those in terms of on-line performance. This reveals the fact that *HCS* is effective in reducing the number of incorrect responses. Since the same kind of selection algorithms were used for both the single-member families and multiple-member families, the main reasons for the large number of wrong responses in single-member families are attributed to genetic disruption and premature convergence. These two problems are related to how strengths are assigned to new classifiers (families) generated by genetic operations since the evolution processes are guided by strengths. In our experiments, the strength of a new classifier is calculated as following:

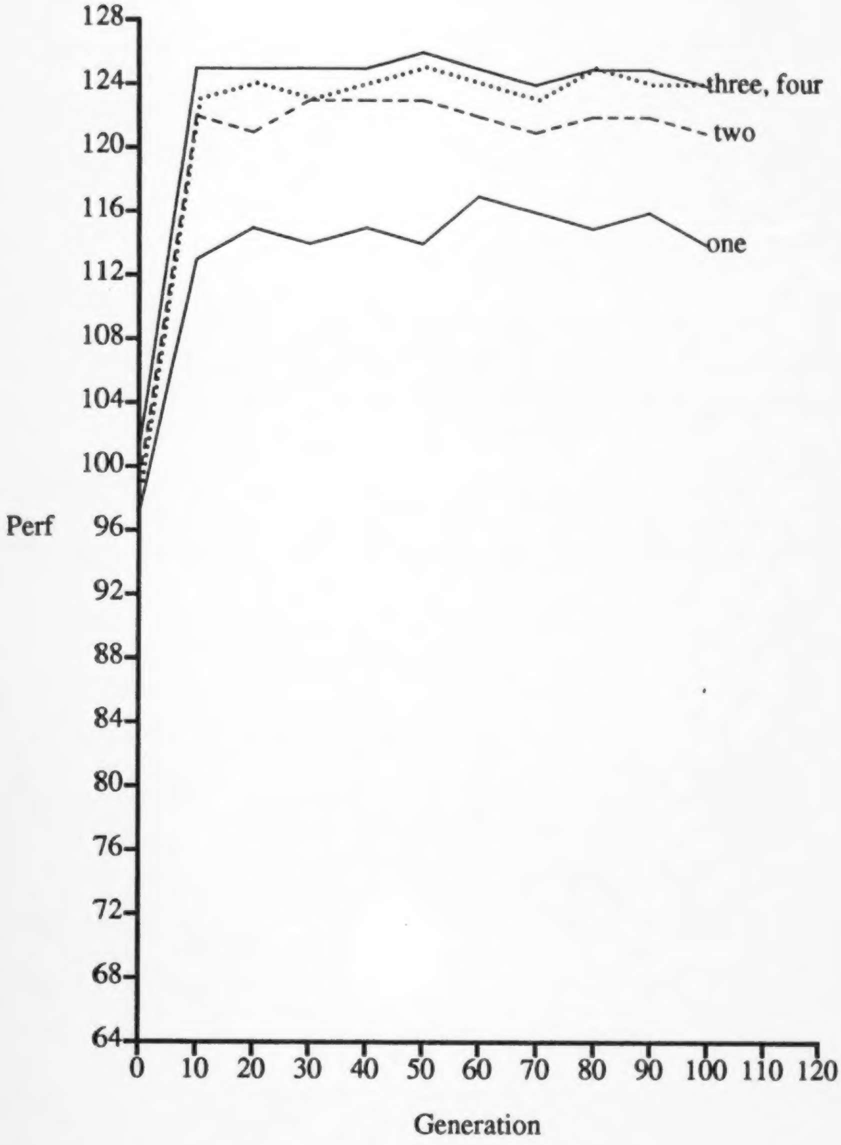

Figure 4.9: f_2 instantaneous performance, ps=240

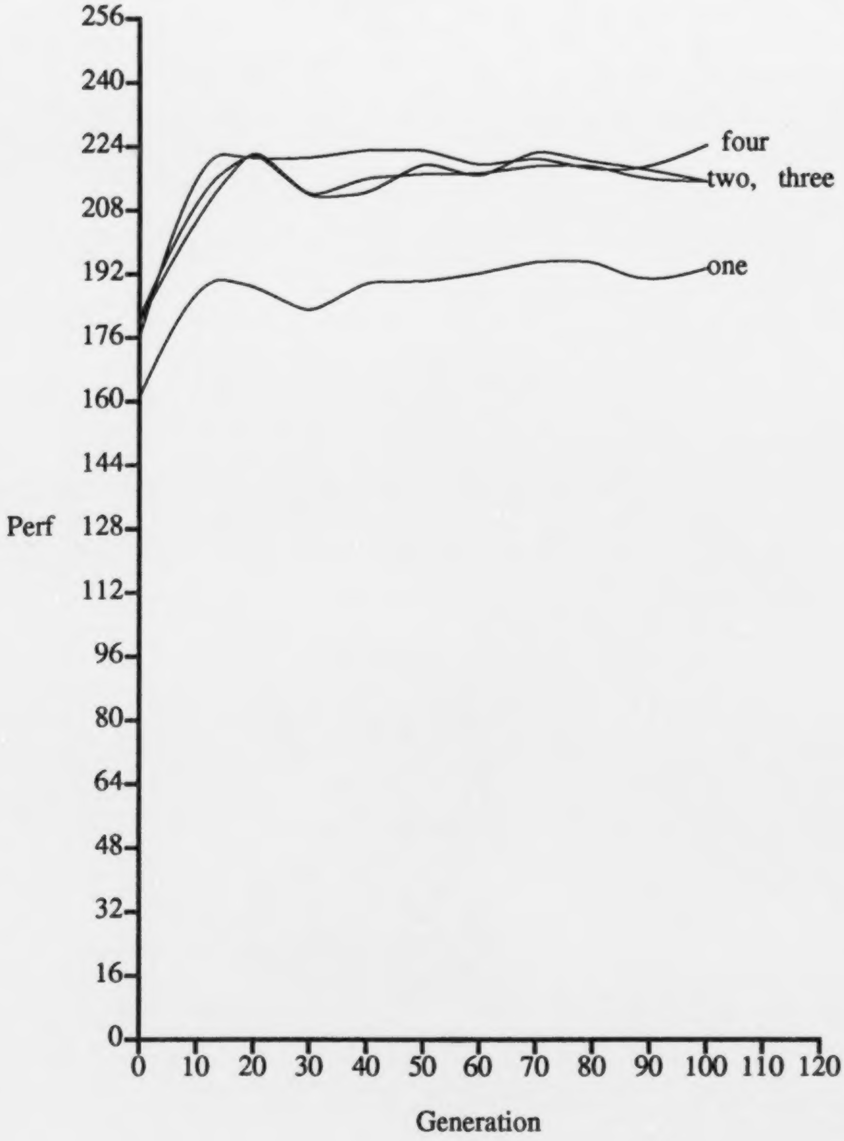

Figure 4.10: f_3 instantaneous performance, ps=360

$$S_{newCla} = (Parent_1\text{'s strength} + Parent_2\text{'s strength}) / (2 \times fs),$$

where $Parent_1$ and $Parent_2$ are the selected families.

The strength of a new family is calculated as:

$$S_{newFam} = \textit{the average strength of the families in the population.}$$

It can be seen that the new strengths are usually at, or above the average (in the case of classifiers). This implies that if a disruption occurs, the disrupted classifiers (or families) would be competitive when competing for firing because of their high strengths. Therefore incorrect responses would be generated by them. With large population sizes, the probability of premature convergence is smaller in both *HCS* and the conventional classifier systems. In this case, genetic disruptions would be a major reason for a poor performance. Differences between the numbers of incorrect responses in *HCS* and conventional classifier systems indicate that the disruptions were reduced in *HCS*. Our experiments with both large and small population sizes (relative to the problems) show that *HCS* is effective in reducing genetic disruption and premature convergence.

Next, we use a typical run for learning function f_1 to show how the rule clustering problem is reduced in *HCS*. A population size of 60 is used in this run. Therefore, in the case of the classifier system ($fs = 1$), there are 60 individuals in the population. In the *HCS* with a family size of 2, there are 30 families. The initial populations for the classifier system and the *HCS* are shown in Appendix 2 and Appendix 3 respectively. All the classifiers have an initial strength of 1. All the families have an initial strength of 2.

At the end of generation 100, the dominating classifiers in the classifier system are:

1. 1###/1
2. #00#/1
3. 0011/1
4. 0##0/1
5. #1#1/0
6. 0111/1
7. ##10/1
8. 1010/0.

The dominating families in the *HCS* are:

1. {##10/1, 1#00/1}
2. {0001/1, #001/0}
3. {1###/1, #0##/0}
4. {#011/0, #000/1}
5. {0011/1, #01#/0}
6. {11##/1, 1111/1}
7. {1010/0, 011#/1}
8. {1###/1, 0111/1}
9. {010#/1, 0#0#/0}
10. {##01/1, 011#/0}.

Let's examine the status of schema ****/0. Notice, in the initial population of the classifier system, there are 34 instances of schema ****/0. But, at the end of generation 100, there are only two instances (classifiers 5 and 8) of schema ****/0 among the dominating classifiers. In the case of *HCS*, there are 34 instances of schema ****/0 in the initial population. After generation 100, there are 7 instances of ****/0 among the dominating families. A solution set for f_1 is

{
00##/1,
11##/1,
##00/1,
##11/1,
0101/0,
0110/0,
1001/0,
1010/0
}.

It is obvious that in the conventional classifier system case, the population is dominated by the pattern ****/1 and the other necessary pattern ****/0 is being replaced by the dominating one. However, in the *HCS*, schemata ****/1 and ****/0 are paired into families, such as families 2, 3, 4, 5, 7, 9 and 10 listed above. Therefore, the fate of the two schemata are bound together by the family ties. This example shows how *HCS* solves or reduces the rule clustering problem.

Experiments were also done with crossover on both families and classifiers in the same runs. Crossing over the columns of the families was used. The results are consistent with the above analyses. Figure 4.11 shows this.

In conclusion, a large majority of the experiments conducted showed no anomalies. *HCS* ($fs \geq 2$) consistently out-performed classifier systems ($fs = 1$). Enhancing classifier systems to support families appears to provide a significant improvement on the performance.

4.4.4. Discussion

HCS was implemented in a simple way. Classifiers are grouped together randomly. Nevertheless, *HCS* shows its abilities of encouraging co-adaptations and maintaining good classifier sets. The main benefit of the structuring lies in its ability to maintain good structures and classifier sets. Once a correct default hierarchy is built in a family, the chance that the hierarchy is well maintained in *HCS* is much higher than in conventional classifier systems. Without specifying a relation over the family members, a system may take a long time to overcome some wrong groupings of family members. For example, classifiers 1##0/1 and 1110/0 form a default hierarchy. But they may be split into two families. If the other members of the families have high utilities, these two classifiers may become parasites and disrupt performance. Specifying a relation r_i as a default hierarchy relation can help solve this problem because then default hierarchies are likely grouped into the same family. Further studies on

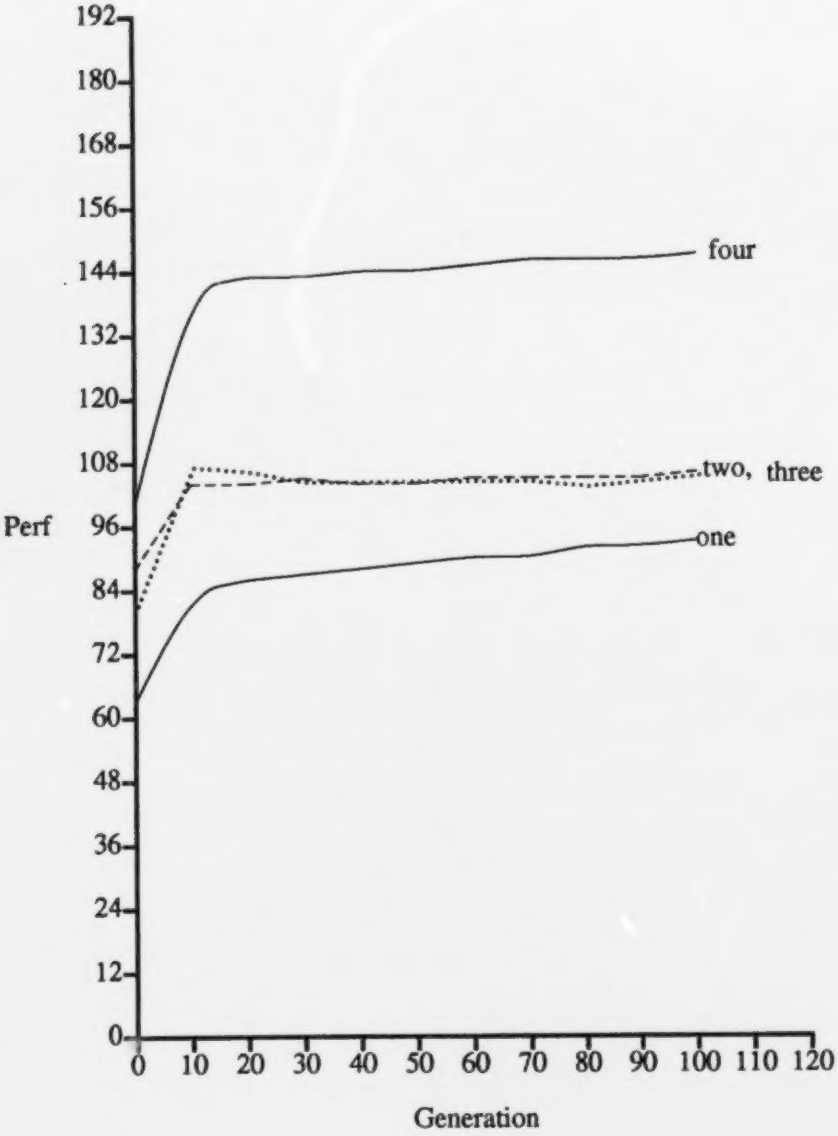

Figure 4.11: f_3 on-line performance, ps = 120

how to form up families according to a relation r_i and how to maintain r_i when conducting the GA operations are needed.

Allowing varied family sizes in a population is another interesting issue. The coexistence of different sized families would be amenable to building families according to relations r_i and help in reducing the parasite problem.

There are only two levels (classifiers and families) in the *HCS* we implemented. For problems with large search and solution spaces, *HCS* can be generalized into a hierarchy with more than two levels. The main motivation for building hierarchies is to gain better organizations. Too many hierarchies could put an extra burden on a system. As in a human society, a small village with only a few families probably does not need to be divided into communities, which is common for cities.

4.5. Towards a Unification of the Michigan and Pittsburgh Methods

Since our goal is to evolve classifier systems which emphasize low-level processing, the previous discussions have been focused on the comparisons between *HCS* and the Michigan type classifier systems [DeJ88], where single classifiers are the basic units for genetic operations. Here, we discuss the relationships between classifier systems, *HCS* and the Pittsburgh approach. In the Pittsburgh systems, the basic units for genetic operations are production system programs (see Sec 1.2). A fitness measure called strength is associated with each program and used to guide the selections for genetic operations. The strength of a program is changed by evaluating the performance of the program on a learning task. The following is an example of a crossover operation on two programs. Assuming

$$R_1 = \{R_{11}, R_{12}, ..., R_{1n}\},$$

and

$$R_2 = \{R_{21}, R_{22}, ..., R_{2m}\},$$

are two programs, where R_{ij} is a production rule. A crossover operation on the two programs may generate the following two new programs:

$$R_1' = \{R_{11}, R_{12}, R_{23}, ..., R_{2m}\},$$

and

$$R_2' = \{R_{21}, R_{22}, R_{13}, ..., R_{1n}\}.$$

The main disadvantage of the Pittsburgh method is its computational complexity in both time and space. As pointed out previously, the search space increases combinatorially with the size of the production program. We use the following example to illustrate the complexity differences between the Pittsburgh approach and *HCS*. Suppose a solution for a task needs four classifiers and each classifier is a string of four bits over the alphabets of {'0', '1', '#'}, then there are totally 3^4 different classifiers. An example of a population of a *HCS* with a family size of 2 may look like:

{000/1, 001/0},
{0#0/1, 00#/#},
.
.
.
{##0/1, 0#1/0},

where classifiers in a pair of brackets constitute a family. The following is an example of what a population in a Pittsburgh system may look like:

000/1 001/0 0#0/1 00#/# 110/#,
001/# 00#/# 011/1 0#0/1 ##1/0,
.
.
.
000/1 001/0 011/1 0##/0 ##1/0,

where each string is 20 bits long if we ignore the '/'s which separate the condition part from the action part in a classifier. In a *HCS* with a family size *fs* of 2, the 4 solution classifiers can be held by two families. Therefore the maximum size of the search space in this case is $\left[\begin{matrix}\left[\begin{matrix}3^4\\2\end{matrix}\right]\\2\end{matrix}\right] = \left[\begin{matrix}3240\\2\end{matrix}\right] = 524718$. With the Pittsburgh approach, the shortest length of each string in the population is 16 bits since the solution contains 4 classifiers and the string that is being searched is supposed to contain at least all the solution classifiers. Hence, the minimum size of the search space in this method is $(3^4)^4 = 3^{16} = 4304672$. One can see, along with the growth of the length of the classifiers and the size of a solution space, the search space in the Pittsburgh approach would be much bigger than that in the *HCS*. Since every string in the Pittsburgh approach is the concatenation of all the classifiers in a complete program that can perform the tasks, the lengths of the strings are usually much longer than that of a single classifier. The population size relative to a search space is limited due to the restriction of resources when strings are too long. Therefore, in the Pittsburgh approach, the percentage of the samples out of an entire search space is much smaller, which tends to cause the problem of premature convergence. The main advantage of the Pittsburgh method over the Michigan method is its ability to maintain rule structures and emphasize the cooperation among classifiers. Compared with the Michigan method, *HCS* provides mechanisms to maintain structures and encourage the cooperation among classifiers. Compared with the Pittsburgh method, *HCS* reduces the computational complexity dramatically by organizing classifiers into small structures instead of complete programs.

However, the difference between the sizes of the structures, i.e. programs in the Pittsburgh approach and families in *HCS*, is not the only characteristic that distinguishes the two methods. There are several fundamental differences between *HCS* and the Pittsburgh method. In *HCS*, a family consists of a set of classifiers. Usually, those classifiers do not constitute a complete program. The idea here is to organize classifiers. This changes the traditional one-level classifier system architecture (classifiers) to the multiple-level architecture (individuals and families). In *HCS*, the credit assignment process is based on the evaluations on individual classifiers. The uses of credit assignment algorithms in *HCS*, are in the same way as those in the conventional Michigan method classifier systems. Families gain credit through the gain of family members. This allows for two levels of competition among classifiers. One is the competition among classifiers in different families. The other is the competition among classifiers in the same family. This illustrates one of the important differences between *HCS* and the Pittsburgh method: *HCS* inherits the implicit parallelism of the Michigan method in the sense that each evaluation of a classifier affects many classifiers and their families. The evaluation process in *HCS* is done through matching and firing individual classifiers, which is similar to what is done in a classifier system. In the Pittsburgh approach the evaluation is done by letting the programs which are the individuals that are being evaluated perform tasks.

In *HCS*, the genetic operations are based on families, which reduces the competition between family members, thus reducing disruptions. *HCS* inherits the computational characteristics of the Michigan method in terms of the implicit parallelism and the lower computational complexity. When the family size is 1, *HCS* is just the Michigan method.

4.6. Remarks on HCS

A method of achieving coherence and cooperation through imposing ties is proposed. Hierarchies are used to impose structural ties in classifier systems, which effectively reduced some fundamental difficulties of classifier systems such as the problems of rule clustering and rule association. Using the hierarchy approach, classifier individuals are grouped into "families". The structural family ties encourage the cooperation among family members, which has the effect of enhancing the stability of a system. The experiments with learning Boolean functions have shown significant improvement in performance and stability of the hierarchical classifier systems over the conventional classifier systems. *HCS* is a model towards a unification of the Michigan and Pittsburgh approaches.

Internal message passing in *HCS* works in the same way as in conventional classifier systems. However, as noted earlier, in the experiments with learning Boolean functions, there is no internal message passing and reasoning chain involved. Although not explored in this thesis, we believe that the structural ties in *HCS* have the additional potential to enhance the stability of chain structures built through the internal message passing. We expect that the family structure will help the establishment and maintenance of chains for the following two reasons.

First, in an internal message passing classifier system with the bucket brigade algorithm, usually there is a payoff delay to the classifiers at the beginning of a chain. That is, the stage-setting classifiers in a chain usually do not get payoff right after they have passed a correct internal message to the message list. The longer is a chain, the longer the payoff delay to the beginning of the chain is. The payoff delays result in the uneven increases of credit (utility) among the classifiers at the beginning of a chain and those at the end of the chain. This produces a possibility for unfair competitions later on, e.g. weaker classifiers (classifiers with less credits) may be deleted from the population. In *HCS*, weaker members in a family are supported by the stronger ones, therefore the adverse effect of the unevenly increased credits among a chain caused by the delayed payoff is much reduced.

Second, if the rules in a chain reside in the same family in *HCS*, the possibility of GA disruptions among the chain is small since the probability of conducting GA operations within a family is small.

In *HCS*, chains can be generated within the family structure as well as across the family structure since the credit assignment algorithm is based on individual rules. But in both cases, the family structure will help the co-adaptation of the chain members. We mentioned in section 4.3 that it is possible to group rules into the family structure by a relation r_i. If r_i is specified as a chain structure, chains would be forced to form within the family structure, therefore reducing the possibility of the chain being disrupted.

Chapter 5

Implicit Semantic Ties Approach to Classifier Systems

5.1. Introduction

In chapter 1, a primitive solution is defined as a classifier that replys correctly to one and only one environmental or internal state. By the definition, a primitive solution does not contain any '#'s. In chapter 4, hierarchies are used to introduce better data structures into classifier systems and therefore strengthen the associations among primitive solutions and improve system performance. This chapter presents a semantic approach for strengthening the associations of the primitive solutions and improving system performance.

In many ways, learning is a process of establishing relations. This requires a learning system to be able to represent the emergence of these relations. For many problems, using variables is an effective way to do this. A variable is a relation defined over its instances. In this sense, a concept is a variable, which represents a general idea, thought or understanding. For instance, "rectangle" is a concept of a group of geometric objects. It can also be considered as a variable that represents figures with four straight sides forming four right angles. Symbol processing is considered fundamentally a process of variable interpretation. A system without variables is limited in its ability of symbolic processing. Lack of variables in neural nets is considered a major deficiency of the approach [Nor86].

Classifier systems use production-rule-like classifier string representations. The advantages of this simple representation have been discussed in chapter 2. However this simple representation also implies the inadequacy of the representational capabilities of the classifier system approach. Under the classifier system framework, primitive solutions are loosely connected to each other. This simplistic representation scheme makes it hard to manipulate knowledge and to add built-in knowledge and models which can be used to describe features common to a problem domain. Compared with conventional symbolic approaches, it is difficult for classifier systems to abstract and explain events. In many cases, classifier systems are incapable of representing information about commonly occurring patterns in a succinct way, such as what frames do in the symbolic approach. It is difficult to build conceptual hierarchies in classifier systems. Here, the conceptual hierarchies refer to those that consist of cognitive level and high-level (symbolic name level) concepts. Consider the example of learning the concept "rectangle". This problem can be described in two ways. One way is to use high-level attributes, such as "equal of two sides" and "not equal of two sides". The other is to use low-level primitive attributes, such as the lengths of the sides. Based on the logic that the relations over primitive data are at a higher level than the primitive data themselves, the first description is at a higher conceptual level than the second one. Conventional classifier systems can perform learning tasks when a problem is described in the first way. In this case, a '0' can be used to represent the "equal" relation and a '1' to represent the "not equal" relation. If a problem is described in the second way, conventional classifier systems would have difficulties. In this case, the relationships among the sides are not among the attributes being represented explicitly. The system needs to build up this relationship based on the primitive data, i.e. the lengths of the sides. However, since the number of the instances of the concept "rectangle" is infinite, in a finite classifier system, there is no way to represent these relationships between the explicitly represented attributes by including all the instances into the population. Therefore, it is very difficult to build up the relation on the primitive data. The above example illustrates the representational difficulty of classifier systems in abstracting relationships.

Variables are fundamental in symbol processing and problem solving. The lack of variables in classifier systems results in the inadequacy of its representational ability, which results in many difficulties in a dynamic process of learning. First, without variables, it is difficult to narrow the gap between classifier systems and conventional symbolic learning approaches in their representational abilities. Hence, it is hard for classifier systems to adopt the merits of symbolic learning, such as built-in knowledge and knowledge models. Second, the problems of rule clustering and rule association are aggravated because primitive solutions or sub-solutions in a classifier system usually are loosely structured. In classifier systems, the only explicit tie for binding primitive solutions is '#'. Hence, many relationships among the primitive solutions can not be represented in a succinct way. Usually, many classifiers would be needed to represent a solution. Therefore, those loosely-connected primitive solutions or sub-solutions tend to suppress each other at early stages of a run and disrupt each other at later stages.

In this chapter, variables are used to introduce better data structures and establish associations between primitive solutions. A new framework, Variable Classifier System (VCS), is introduced in which classifier systems are evolved to include variables. Conditions and actions in rules (classifiers) are broken into fields, each of which can be a constant or a named variable. The result is that variables act as implicit ties between the instances being represented by them and thus allow some problems to have their solutions expressed in a succinct manner, reducing the amount of work required by genetic search to solve a problem. This approach alleviates the problems of rule clustering and rule association. As well, variables provide a way of describing an abstract world, allowing for building of models and knowledge structures as in high-level symbolic representation systems. Conceptual hierarchies can be more easily built in a variable classifier system. It thus becomes possible to combine the advantages of high-level and low-level representations into one framework. The experiments show the expected results. When the connection degree of primitive solutions is increased by variables, VCS shows significant improvements over the conventional classifier system. The problems of rule clustering and rule association are shown to be reduced effectively.

In the following sections of this chapter, the design of VCS framework is given first. Then the properties of VCS are discussed followed by the experimental results and analyses. Finally, problems in VCS are discussed followed by concluding remarks.

5.2. VCS Framework

Similar to a classifier system, VCS is a low-level representation framework in which the idea of variable is introduced. VCS retains the structure of classifier systems. There are message and rule lists and, in each cycle, messages match rule conditions generating new messages from matching actions. Genetic operators and the bucket brigade operate as in CS. Rules can be negated and have multiple conditions. The difference between VCS and CS lies in the syntax and semantics of messages and rules.

The basic symbols in VCS are the alphabet {'0', '1', '?', '*'} and a set of logical field placeholders. The question mark '?' sign replaces the '#' in classifier systems and represents either don't care or don't know situation. The "don't know" situation can be used as a prompt for the user to provide some information. This allows a system to communicate with the external world by providing useful feedback and prompting for useful information or instructions. The symbols '0' and '1' in a condition can be matched only by '0' and '1' in a message respectively. The '?' in a condition can be matched by '0', '1' or '?'. A '?' in a message can only be matched by '?' in a condition. The '?' here and '#' in classifier systems are semantically and functionally different. In addition to the meaning of "do not care" (equivalent to '#'), '?' also has the meaning of "do not know". This makes the functional differences between VCS and CS. With the "do not know" meaning of '?', VCS is able to communicate with its environment, providing information to and asking information from the environment. This changed the situation in classifier systems where a system can change its environment only by imposing actions onto the environment. No changes happen to an environment to meet some dynamic requirements of the performance of a system.

Messages, conditions, and actions are implemented as fixed length bit strings consisting of a number of fields. Each field represents an attribute of the problem domain. To distinguish the implementation (strings of '1's and '0's) from the semantics, field identification placeholders are inserted in messages, conditions, and actions to identify the contents of each field. For readability, we use the notation $P_{contents}$ to name a field, where the identifier *contents* is meant as a meaningful interpretation of the semantics of that field. For brevity, we often name fields numerically $\{P_1, P_2, P_3, ..., P_n\}$. For example, consider the blocks world, an oft-cited example that many problem solving programs use. In the VCS representation, a rule condition could consist of two fields:

P_{name}: the name of a block

$P_{relation}$: relations between this block and other blocks

and the condition could be represented as

P_{name} 10 $P_{relation}$ 0110110

where "10" is the name of a block and "0110110" represents its relationships with other blocks. Removing the field names yields a familiar classifier system notation.

As mentioned previously, the use of '?' introduced the important interactive property into VCS. Another major difference between VCS and classifier systems is the inclusion of the '*' operator into rule conditions and actions (but not messages). The '*' character in a field (anywhere in the field) indicates that the content of the field should not be interpreted as a constant but as a variable. For example, a 3-bit field could contain "*01", "00*", or "*?*", any of which indicates that the field is to be treated as a variable. In the following discussion, we will separate the semantics of a representation from its implementation. While using the {'0', '1', '?', '*'} notation for the implementation, we prefer to use names for our variables, such as x, y, and z (instead of "*01", "00*", "*?*") in our discussion to increase the clarity. Using the block world example again, the condition

$$P_{name}\ x\ P_{relation}\ y$$

indicates that the two fields are variables. For example, the condition

$$P_1\ {*00}\ P_2\ ?1\ P_3\ 0{*}{*}0\ P_4\ {*00}$$

is equivalent to

$$P_1\ x\ P_2\ ?1\ P_3\ y\ P_4\ x$$

where P_1's x is independent of P_4's x.

In VCS, the semantics of matching messages with rule conditions are different from CS. A condition field containing only {'0', '1', '?'} is treated as a constant and matches the corresponding field in a message if all the characters match (recalling the slightly different semantics of '?' mentioned above). A condition field that is a variable matches the corresponding field in the message, ***regardless of the content of that field.*** As such, we view each variable field as being a ***parameter*** of the rule; matching then becomes binding of values (from the message) to parameters (conditions of the rule).

A user of VCS must define his messages in terms of the number of fields that constitute a classifier and the size of each field. Fields of the same type, i.e. with the same name, can appear in a classifier more than once. But the sizes of these fields must be the same. Fields with the same name have the ability to exchange information between them through variables. If the same variable appears in a classifier more than once, then all those appearances must be bound to the same value in all cases, otherwise the matching fails (see example below). This feature allows the equality relationship to hold between fields in (multiple) conditions.

Variables can also appear as part of the action of a rule. Such variables take their values from those assigned in the condition. If a variable does not appear in the condition part, the variable in the action part will randomly take a value from the value domain of the field. Consider the rule in which '/' is used to separate condition from action

$$P_1\, 11\, P_2 *00\, P_2 *01 \,/\, P_1\, 01\, P_2 *01\, P_2 *00$$

with semantics

$$P_1\, 11\, P_2\, x\; P_2\, y \,/\, P_1\, 01\, P_2\, y\; P_2\, x\ .$$

Then the message

$$P_1\, 11\, P_2\, 101\, P_2\, 111$$

will match the rule, and result in the information exchange between the last two fields. The resulting message will be

$$P_1\, 01\, P_2\, 111\, P_2\, 101\ .$$

Note that information exchange can only occur between the second and third fields since they have the same name. The exchange of values between fields is not possible in classifier systems, without enumerating all the possibilities.

When using variables, some rules with different appearances may have the same semantics. For example the rules

if v_i and v_{i+1} then v_{i+2}

if v_j and v_{j+1} then v_{j+2}

have the same meaning. To avoid this, our system enforces a *normalized* representation for rules. In this representation, each rule must use the minimum number of variables possible, and the variable names must be ordered and sequentially starting from a fixed name (e.g., $v_1, v_2, v_3, \ldots$). We say that the order of v_{i+1} is higher than the order of v_i. In a rule, a higher order variable name for a parameter can be used only after all the lower order variables of the same parameter have been used. Under this policy,

$P_1 v_1 P_1 v_2 / P_1 v_2 P_1 v_1$

is legal. But

$P_1 v_2 P_1 v_1 / P_1 v_1 P_1 v_2$

is not since $P_1 v_2$ precedes $P_1 v_1$. As a result of this enforcement, there are many illegal rule combinations. The *semantics* of VCS insist that semantically illegal rules with respect to variable usage are not allowed to be created. This greatly reduces the search space of possible rules.

One last note on variables. A field may not be defined large enough to represent all the possible variables of the field. For example, if a field is only 1 bit long, one cannot represent the two different values of this field by using two different variables. Such a field can represent only one variable, ‘*’. VCS recognizes this problem and expands fields, if necessary, so they can properly accommodate all possible values.

5.3. Properties of VCS

VCS keeps the low-level representation property which is vital to low-level processing. Any point in a problem space can be represented by a permutation over a set of primitives. VCS greatly enhances the representation capability of conventional classifier systems. As a result, some problems of classifier systems, such as rule clustering and rule association, are effectively reduced. As well, VCS provides a model that combines the advantages of low-level and high-level processing.

5.3.1. Uniform Representation

VCS inherits the representational advantages of classifier systems. Every state of a problem space can be generated by the genetic operators due to the uniform representation, i.e. every point in the problem space is represented by a permutation over the primitives of {'0', '1', '?', '*'}. This uniform representation enables VCS to avoid constructing new rules grammatically, thus simplifying the process of generating new rules (classifiers). As in conventional classifier systems, VCS representation supports the important property of implicit parallelism.

5.3.2. Expressiveness

VCS can be considered as a general parameter representation framework. To use the framework, the designers need only decide what fields are required to describe their problems (as in CS). For example, in the "rectangle" problem mentioned previously in this chapter, the lengths of a geometric object may be used to describe the problem, or at a higher level, the relationships between the sides of an object could be the parameters used to describe the problem states. Compared with CS, VCS is more expressive. One obvious example is that it is difficult for a classifier system to represent relationships among fields without enumerating all possibilities. For example, the symmetric relation *if R(x, y) then R(y, x)* and the transition relation *if R(x, y) and R(y, z) then R(x, z)*. In classifier system, the above relations cannot be represented by single classifiers because there is no way to distinguish the semantics of the '#' signs. But, obviously, these relations can be represented in VCS. For example, the relation *if R(x, y) then R(y, x)* could be represented in VCS as:

$$\begin{array}{cccccccc} P_r\,11 & P_a\,{*}01 & P_a\,{*}00 & \text{--->} & P_r\,11 & P_a\,{*}00 & P_a\,{*}01 & \\ \mathrm{R} & \mathrm{x} & \mathrm{y} & \text{--->} & \mathrm{R} & \mathrm{y} & \mathrm{x} & . \end{array}$$

However, classifier ### ### ---> ### ### may be the best representation for this in CS. By using multiple conditions in VCS (separated by commas), the transitive relation could be represented as

$$P_1\,x\;P_1\,y,\,P_1\,y\;P_1\,z\;/\;P_1\,x\;P_1\,z\quad .$$

In classifier systems, there is no obvious way to enforce equality of fields between conditions.

5.3.3. Search Space

Classifier systems use strings over the alphabet {'0', '1', '#'} yielding a search space of size 3^l, given l characters in a message (without losing generality, we ignore the action part of a classifier). In VCS, the alphabet consists of {'0', '1', '?', '*'}, yielding an upper bound of 4^l on the search space †. Normalizing the representation greatly reduces the number of semantically correct possibilities.

What is a bound on the search space, given normalized and semantically correct rules? An upper bound on the average search space is $S_{VCS} = 3^l + N_n \times 3^{ma}$, where N_n is the number of all the legal strings that contain variables, n is the number of fields, m is the average length of the fields, and a is the average number of fields that are constants in a legal string that contains variables. Given n fields, it is possible to have up to n variables in a rule. We have $N_n = f_{(n,0)} + f_{(n,1)} + \dots + f_{(n,n)}$, where $f_{(n,0)}$ is the number of strings with n fields that has no variable name in the last field and $f_{(n,i)}$ is the number of strings containing variables and having variable v_i in the last field. We have

---

† Usually the 'l' in VCS is the same as the 'l' in CS for the same problem. However, when the length of a field in CS is 1, the number of characters in a message in VCS is twice of the number of characters in a message in CS since the shortest length of a field in VCS is 2.

$$N_n = f_{(n,0)} + \dots f_{(n,n)} \, ,$$

$$f_{(1,0)} = 1,$$

$$f_{(1,1)} = 1,$$

$$f_{(n,i)} = \sum_{j=i-1}^{n-1} f_{((n-1),j)}, \quad i = 1, \dots, n \, ,$$

$$f_{(n,0)} = \sum_{j=0}^{n-1} f_{((n-1),j)}, \quad n = 1, 2, 3, \cdots \; .$$

Table 1 gives some values of 3^l, 4^l, N_n and $(3^l + N_n \times 3^{ma})$ when $l = 10$ and $a = n / 2$. For those practical values of n, m and l ($l = n \times m$), S_{VCS} is much closer to 3^l than 4^l. Since the minimum length of a field is 2 in a VCS, the maximum value for the number of fields, n, is 5 given $l = 10$.

As in CS, a string in VCS is a representative of many schemata. Hence VCS still retains the property of implicit parallelism of classifier systems.

Using multiple conditions in a classifier has the potential to reduce the search space for some problems. In a fixed length, single condition CS representation, the condition string must include all information necessary for solving a problem. For example, a CS representation for the game of tic-tac-toe might include all 9 squares on the board [Sch88]. However, VCS allows relationships to hold between multiple conditions. This permits smaller messages to be used, using multiple conditions to bind them together. In the tic-tac-toe case, one could have each message representing the contents of one square. Then using three conditions for a rule, one can define ***in a single rule*** the information that ***three in a row wins***. This is not possible in classifier systems.

3^{10}	n	N_n	$S_{vcs} = 3^{10} + N_n \times 3^{ma}$	4^{10}
59049	1	2	59535	1048576
59049	2	5	60264	1048576
59049	3	14	62451	1048576
59049	4	42	69255	1048576
59049	5	132	91125	1048576

Table 1.

Consider another example. The task is to manage a terminal room according to terminal availability and the identity of the party making request. We can use 2-bit strings to represent the state of a terminal: occupied = 00, available = 01, and damaged = 10. Assuming there are 6 terminals, we use 3-bit strings to represent the terminal names: terminal 1 = 001, terminal 2 = 010, etc. Also assuming there are 6 parties involved, we use 3-bit strings to represent the identity of the parties: party 1 = 001, party 2 = 010, etc. We want to represent a usage constraint:

> *if terminal x is available and party x is requesting*
>
> *then open the door.*

We use a terminal's name, the terminal's state and the requesting party's identity as the constraint explicitly showed in the condition part. Under such a scheme, the CS representation needs at least two classifiers for each terminal. For example, the two classifiers for terminal 1 are:

if terminal 1 is available and party 1 is requesting
then open the door.

if terminal 1 is not available
or if the requesting party is not party 1
then do not open the door.

If we use 1 for *open* and 0 for *not-open*, the binary format of the representation is

00101001 / 1
00100### / 0.

At least 2*N* classifiers are needed to represent the usage constraint for all the terminals, where *N* is the number of terminals and the number of the parties involved.

In the VCS representation, the above constraint can be represented as:

$P_{name}\ x\ P_{state}$ 01, $P_{name}\ x$ / open

$NOT(P_{name}\ x\ P_{state}$ 01, $P_{name}\ x)$ / not-open

In contrast with CS, the solution space of VCS consists of only *two* rules.

5.3.4. Register Property

When genetic algorithms are used as tools of function optimization, no message passing is involved. When applied to learning, message passing is an important factor that influences both the effectiveness and the efficiency of a system. The message passing scheme of CS is not effective in some situations (see the following examples). The effect of the field identifications and variables in VCS provides the system with a register ability: a value stored in a variable can be used in a variety of places. The following examples show that in some situations, VCS can easily do what CS has problems with.

Example 1

Variables allow values to move to different fields. Consider the rule

if x is on y then y is below x.

There is no simple way of representing this relation in CS with a single classifier. The VCS solution for the above problem is:

P_1 01 P_2 *0 P_2 *1 / P_1 10 P_2 *1 P_2 *0

(on) x y / (below) y x

with the values of x and y in the condition part being passed correctly to the action part.

Example 2

In CS, message passing can cause a performance bias. Consider the rule

if object is small and moving fast
then slow-down or ignore

A classifier representation might look like this:

1 1 / 1 #

(small) (fast) / (tag) (slow-down or ignore)

Only message "11" can fire this rule. Through message passing, the '#' always gets value '1'. If value '0' means "ignore", then this case will never be chosen. The VCS solution for this rule is

P_1 10 P_2 10 / P_3 *1 P_1 11

(small) (fast) / (slow-down or ignore) (misc)

The variable P_3 *1 would be bound to either "slow-down" or "ignore".

Consider the rule

if object is small or big

then slow-down or stop

In a classifier system, this could be represented as:

#   1 /   #   1

(small or big) (fast) / (slow or stop) (misc)

Under the classifier system message passing rules, this equates to:

0   1 / 0   1

(small) (fast) / (slow) (misc)

and

1   1 / 1   1

(big) (fast) / (stop) (misc)

which is not the original meaning.

Although techniques in classifier systems, such as tagging, may solve some of the above problems, two of the side-effects of these techniques are a larger search space and a bigger solution set.

With the register capability in VCS, we can group some properties together to form concepts. By assigning and changing the range of a field P_i, we actually changed the concept P_i. Also, the register property allows equations to be expressed in VCS.

The VCS framework has some other potentials for inductive learning. For example, by using a new identification sign P_{n+1} or by augmenting the value field of P_i, the problem space can be augmented. Thus, a four-block world can be enlarged to a five-block world without much change. This point is also illustrated by the example about allocating terminals, which is used previously in this chapter. In that example, the number of terminals considered in a problem is not a factor that affects the VCS representation. Thus, knowledge gained in a six-terminal problem can be applied to a seven-terminal problem without any change.

5.3.5. Build Structures by Building Abstract Relations

In classifier systems, existing knowledge can be incorporated into a system by an initial classifier pool or by the input messages. However, it is hard to incorporate into these systems the relations among the knowledge components. Structural knowledge cannot be easily built in and manipulated. The strength of the structural knowledge representations lies in the ability to represent relations among the knowledge components through the representation structure itself, providing an efficient way to manipulate these relations.

Frames and semantic networks are two popular structural knowledge representations. Super-ordinate (*super*) and sub-ordinate (*sub*) are two relations that appear in these high-level structural representations. Also, they are the relations among different levels of default hierarchies in classifier systems. These relations are partial orderings on their domains. Therefore, for any X, Y and Z in such a domain, the following rules are true:

if X_{super} Y and Y_{super} Z then X_{super} Z
if X_{sub} Y and Y_{sub} Z then X_{sub} Z
if X_{sub} Y then Y_{super} X

By including such rules in a rule base, we can build a knowledge structure into the system. By using abstract relations, we link the corresponding knowledge together. Then in default hierarchy situations, knowledge can be triggered in the same way as using a high-level structural representation. Notice that the tagging method of representing inheritance in Belew and Forrest's work [BeF88] cannot activate concepts with subordinate and super-ordinate relations in two directions at same time. This implies that the tagging classifier representation of the semantic network is not equivalent to the high-level semantic network representation. However, by adding in the above three relation model rules, the representation is equivalent to the semantic network representation.

5.4. Implementation of VCS

To simplify the implementation, '#' is used instead of '?'. Two kinds of bidding strategies are used. The first one is

$$bid = \frac{k \times u_c \times sp}{2^{l-sp}},$$

where k is a constant, u_c is the utility of the classifier, l is the length of the condition part of the classifier and sp (specificity) is the number of the 'non-#'s in the non-variable fields of the condition part of the classifier. This bidding strategy gives more specific classifiers priority over more general ones and supposedly supports the emergence and maintenance of default hierarchy structures. The second one does not consider the factor of specificity and is defined as

$$bid = \frac{k \times u_c}{l},$$

where k, u_c and l are the same as above. In our experiments, only one classifier is activated at a time. When a conflict occurs, the matched classifier with the highest bid is chosen to be activated and pays its bid to the environment. The action part of the activated classifier checks against with the environmental feedback. If the action matches the feedback, the classifier earns positive payment from the environment, otherwise the classifier earns negative payment. Since learning Boolean functions is used as the learning tasks (see section 5.4.1), classifier chains are not required for solving the test problems. Therefore, internal messages are not used. At the end of each match-activation cycle, a new environmental message is generated and placed onto the message list.

Classifiers without '#'s and variables are stored in an ordered binary tree structure. Thus, the time spent on matching these classifiers is $O(\log m)$, where m is the number of the classifiers stored in the tree. This decreases the time for matching from $O(m)$ when the classifiers are not organized into a tree structure.

The "Roulette wheel" algorithm [Gol89a] is used to select candidates for genetic operations. That is, the probability of a classifier being selected is in direct proportion to its strength (utility). In each generation, the classifier with the lowest utility is replaced by a new one.

In the implementation of VCS, the names of the fields are not represented explicitly in classifiers. The length of a classifier is determined by the number of fields in the classifier and the length of the fields. To represent more than one variable, the length of a field should be at least 2 bits long.

5.4.1. Test Problems

Learning Boolean function is used as the test-bed for VCS. There are two considerations for doing so. First the clear definitions of Boolean functions make it easy to design the syntactic and semantic relationships among different classifiers and different fields of a classifier, therefore to see the advantages or disadvantages of using variables. Second, learning Boolean functions can be considered as a general task, since many tasks can be abstracted to a task of this kind. Several kinds of functions are used to test different facets of variables. The sample functions discussed here are:

$$f_1(x_0, x_1, x_2, x_3) = (x_0 \ AND \ x_1) \ OR \ (x_2 \ AND \ x_3) \ OR \ (NOT(x_0) \ AND \ NOT(x_1)) \ OR \ (NOT(x_2) \ AND \ NOT(x_3)),$$

$$f_2(x_0, x_1, x_2, x_3, x_4, x_5) = x_0 \textit{ AND } x_1 \textit{ AND } x_2 \textit{ AND } x_3 \textit{ AND } x_4 \textit{ AND } x_5,$$

f_3 is an 8 variable function (see appendix for definition), and

$$\begin{aligned} f_4(x_0, x_1, x_2, x_3, x_4, x_5) = {} & (x_0 \textit{ AND } \textit{NOT}(x_1) \textit{ AND } \textit{NOT}(x_2) \textit{ AND } \textit{NOT}(x_3) \textit{ AND } \textit{NOT}(x_4) \textit{ AND } \textit{NOT}(x_5) \\ & \textit{OR } (x_0 \textit{ AND } \textit{NOT}(x_1) \textit{ AND } x_2 \textit{ AND } \textit{NOT}(x_3) \textit{ AND } x_4 \textit{ AND } \textit{NOT}(x_5) \\ & \textit{OR } (x_0 \textit{ AND } x_1 \textit{ AND } \textit{NOT}(x_2) \textit{ AND } \textit{NOT}(x_3) \textit{ AND } \textit{NOT}(x_4) \textit{ AND } x_5) \\ & \textit{OR } (x_0 \textit{ AND } x_1 \textit{ AND } x_2 \textit{ AND } \textit{NOT}(x_3) \textit{ AND } x_4 \textit{ AND } x_5). \end{aligned}$$

f_1 is a function for which VCS has obvious representational benefits. The use of variables changed the structure of the primitive solutions or the structure of the results of learning this function and strengthened the association among the primitive solution classifiers. A smallest solution set for f_1 in CS is { 00##/1, 11##/1, ##00/1, ##11/1, ####/0}. However in VCS, the classifiers in this set are connected in a different way and represented as {*0*0####/1, ####*0*0/1, ########/0} assuming the length of a field is 2. The length of the classifiers is doubled in VCS. But the VCS solution set is more succinct, which is more important for a better performance when a system is suffering from the rule clustering problem.

f_2 is a function for which VCS does not provide any advantages over CS. The connection degree of the primitive solutions is not increased in VCS. The smallest solution set for f_2 in CS is {111111/1, ######/0}. In this case, instead of being advantageous, VCS's performance may be slower because of its larger search space.

f_3 is a minimal function of eight variables and is the sum of 58 productions. So the minimal size of a solution set for this function in CS is 59, of which 58 classifiers are for the value of 1 and one classifier ########/0 is for value 0. Variables are capable of increasing connections of the primitive solutions for this function. For example, 00101#10/1 and 10111#10/1 are two classifiers in the CS solution set. In VCS, they can be represented by one classifier *00011*011##1100/1. f_3 is used to illustrate the difference between VCS and CS when the problem is more complicated and the representation advantage of VCS is not as obvious as in the case of f_2 kind functions.

f_4 is different from the above functions. Representationally, the solution set in VCS is smaller than that in CS. The connections between the primitive solutions are increased in VCS. The smallest solution set in CS is {10000/1, 101010/1, 110001/1, 111011/1, ######/0}, whereas the smallest solution set in VCS is {11*0*100*1*0/1, ############/0}. However, semantically, there are only 4 situations out of 64 in which the value of f_4 is 1. This means the rule clustering problem would not appear to be outstanding since even if the population converges to ######/0, the performance is still over 90%.

5.4.2. Experiment Results and Analyses

The performance of VCS has been examined with different population sizes and compared with the conventional classifier systems. In the following figures which show the experimental results, the horizontal axis measures the number of generations elapsed. The vertical scale represents the performance which is defined as

$$Perf = \frac{\sum_{t=1}^{T}(C_t - I_t)}{T},$$

where C_t is the number of the correct responses and I_t is the number of the incorrect responses in generation t. Each performance line in the figures shows the average of five runs generated with different random number seeds.

In most of the experiments, VCS is examined with only one bidding strategy, $bid = k \times u_c / l$, where k is a constant, u_c is the utility of the classifier and l is the length of the condition part of the classifier. Two bidding strategies are used when examining the performances of the conventional classifier systems (CS). There are two considerations for doing so. First, in VCS, it is not as easy as in CS to determine the value of the specificity factor (*sp*) of each classifier. In CS, *sp* equals to the number of 'non-#'s in the condition part of a classifier. However, in VCS, classifiers with the same number of '#'s may not be at the same level of specificity or generality. For example, classifiers ##01/1 and *xy*01/1 both contain two "non-#"s. But ##01/1 is more general than *xy*01/1. The second reason for this is that we want to verify the hypothesis that VCS does not rely on complicated default hierarchies because of its higher representational "resolution". Therefore VCS would avoid the difficulties of building and maintaining structures such as default hierarchies. We suggest that a performance that relies on default hierarchies would not achieve good results with the bidding strategy without the specificity factor.

In the following figures, the performance of VCS is denoted by ***Wvariable***, meaning "with variables". The performances of CS are represented by ***Nvariable***, meaning "no variables", and ***NvariableS***, meaning "no variables but with the specificity factor considered". The population sizes used in the figures are the representitives of groups of population sizes used in each experiment.

The experimental results illustrate the following points:

1. VCS helps reduce the dependency of a system on loosely connected internal structures, such as the default hierarchy structure. This is illustrated by the following example. At the end of a process of learning function f_1, in VCS, classifier *0*1**1*/0 is one of the dominating classifiers in the population. The sementics of this classifier is that if the first field is not the same as the second field and the third field is not the same as the fourth one, $f_1(X) = 0$. This single classifier covers all the situations in which the value of f_1 is 0. However, the same generation of the population in the conventional classifier system has to use

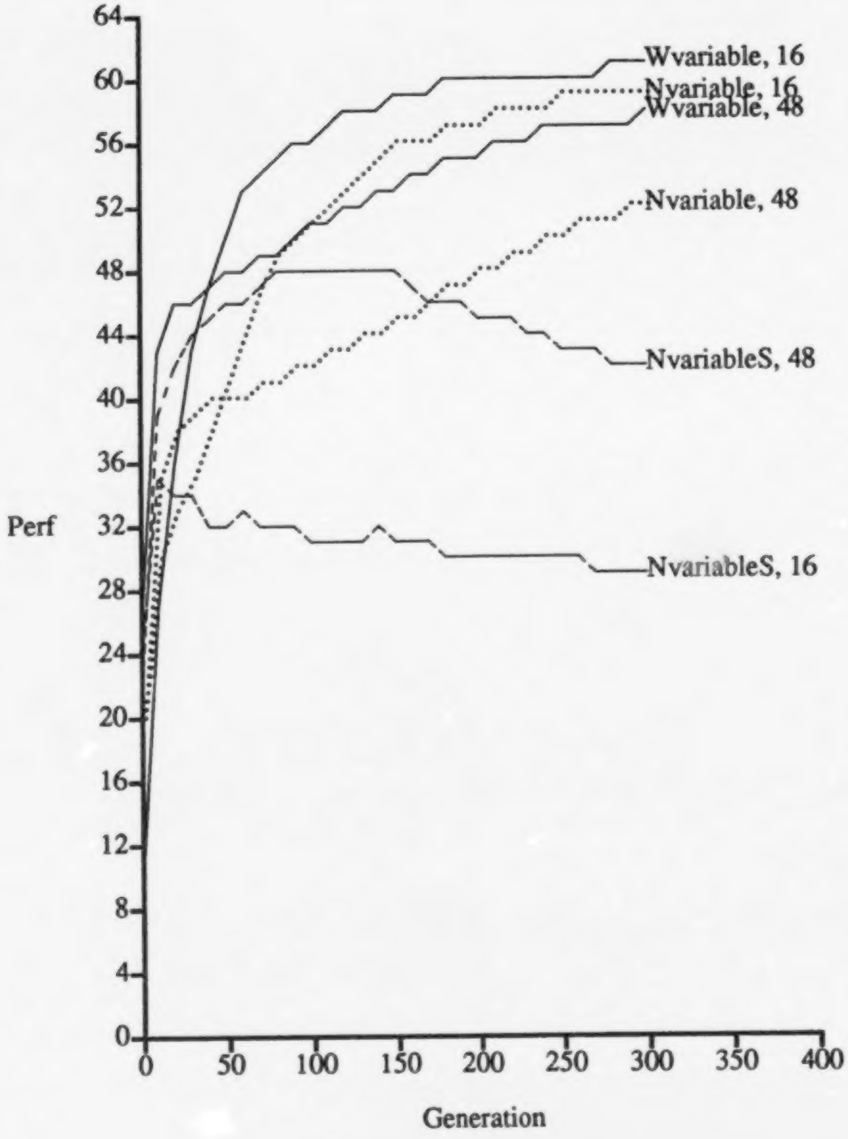

Figure 5.1: f_1 performance

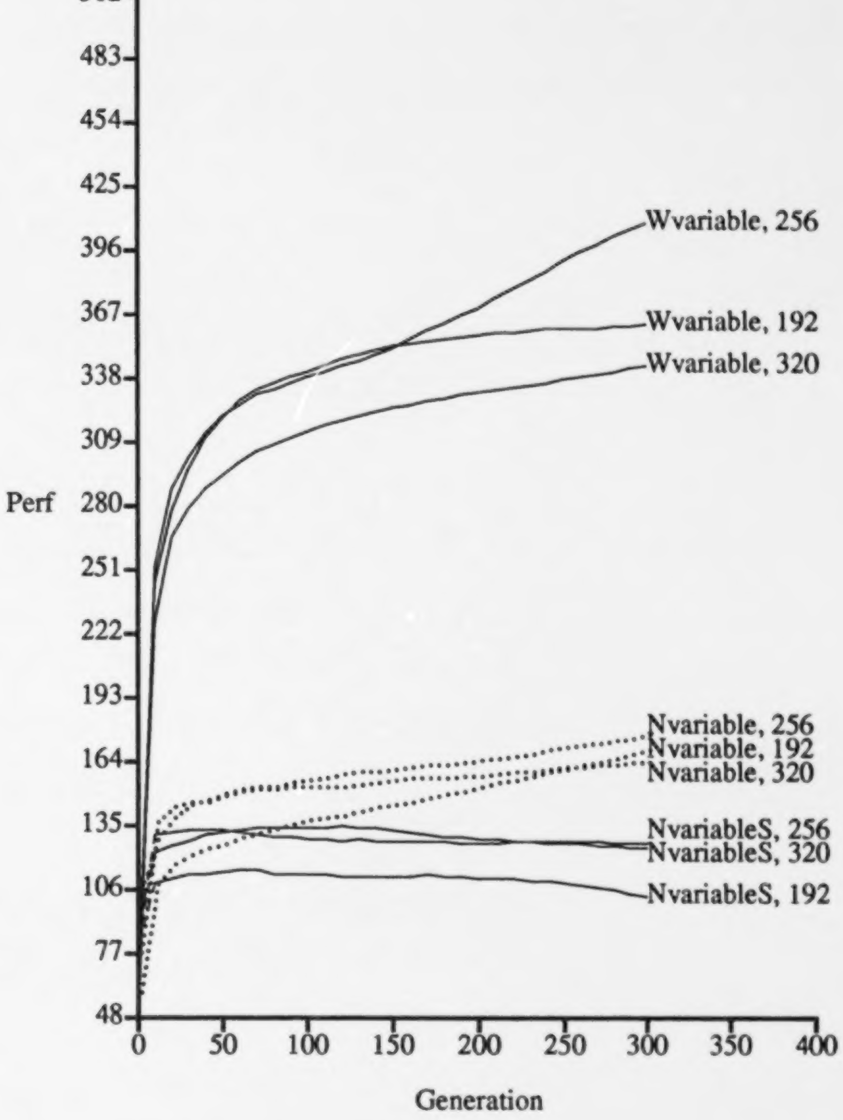

Figure 5.2: f_3 performance

the following classifiers to cover the situations in which the value of f_1 is 0:

1. 0#1#/0
2. 001#/1
3. 0#11/1
4. #001/0
5. 0001/1
6. 1#01/0
7. 1101/1

Let's examine how the above classifier set would react to an $X = (x_0, x_1, x_2, x_3)$, where $f_1(X) = 0$. Notice that classifiers 1, 2 and 3 form a default hierarchy in which classifier 1 is the general classifier that reacts to the conditions 0010, 0011, 0110 and 0111, and classifiers 2 and 3 are the exceptions that react to the conditions 0010, 0011 and 0111 respectively. For classifier 1 to work correctly, the default hierarchy relationship between the general classifier and the exception classifiers must be established and maintained. Classifiers 4 and 6 are in a similar situation to classifier 1. For them to work correctly, classifiers 5 and 7 must work as the exception classifiers respectively. This example shows how the succinct representation in VCS can avoid the difficulties involved in establishing and maintaining internal relationships among classifiers.

2. For problems for which VCS has representational benefits, VCS outperforms CS in both *Nvariable* and *NvariableS*. The experimental results for f_1 and f_3 are used to show this point. Figures 5.1 and 5.2 show two sets of representative results. Figure 5.1 shows the performances for f_1 with the population sizes of 16 and 48 respectively. Figure 5.2 presents the performances for f_3 with the population sizes equal to 192, 256 and 320.

3. In a certain range of population sizes, the performance of VCS increases as the population size (ps) grows. This is shown in Figure 5.2, in which the *Wvariable* with $ps = 256$ is better than that with $ps = 192$. But when $ps = 320$, the

performance slowed down, which implies that if *ps* is too large, the efficiency would be degraded.

4. In a larger range of *ps*, *NvariableS* increases as *ps* grows. This is shown in Figure 5.3. In the case of *NvariableS*, specific classifiers are given priority in the process of conflict resolution (select matched classifier for activation). Therefore, when population size is big enough, the population would converge to classifiers not containing '#'. This is equivalent to using a system without '#' and use one classifier for each possible environmental state. An obvious disadvantage of this type system is its ineffectiveness when dealing with more complicated and larger problems, particularly, problems with infinite environmental states.

5. Another point shown by the experiments is that if variables cannot change the structure of the primitive solutions, i.e. classifiers with variables are not any better than those without variables in representing the solution, the performance of VCS does not show much differences from those of CS. The performance results for f_2 are used to illustrate this point. f_2 is a function for which VCS does not have any representation advantages. Figures 5.4 and 5.5 show the results when $ps = 64$ and 128 respectively. It is worth to point out that in the experiments with f_2, when *ps* is large, VCS did not show any inferiority to CS although it has a larger search space.

6. The results of f_4 are used to show another point. f_4 is a function for which VCS can change the structure of the primitive solutions in a positive way. However, there is only one situation that this change is beneficial, i.e. when 11*0*100*1*0/1 is included in the population. Therefore, when learning f_4, VCS did not show off its advantages in most situations. Figure 5.6 shows a case when VCS outperformed CS. Figure 5.7 shows a typical result.

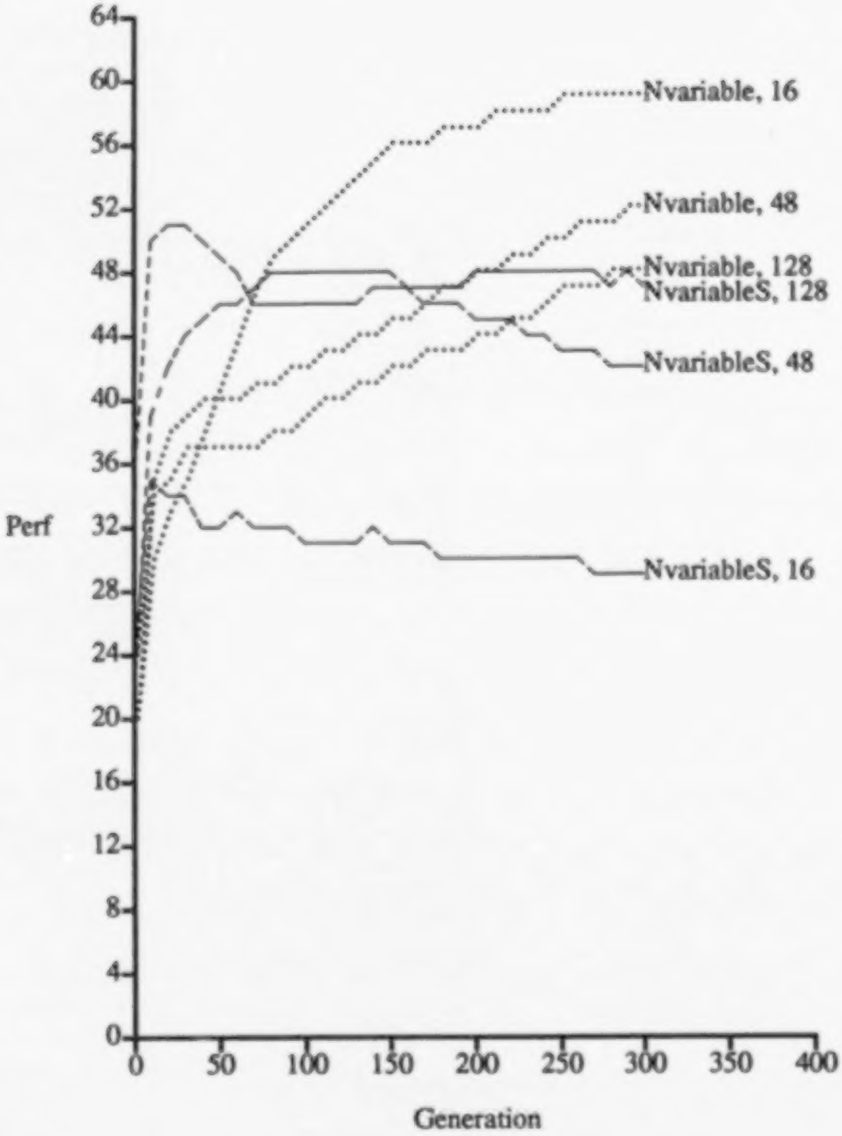

Figure 5.3: *Nvariable* and *NvariableS* of f_1

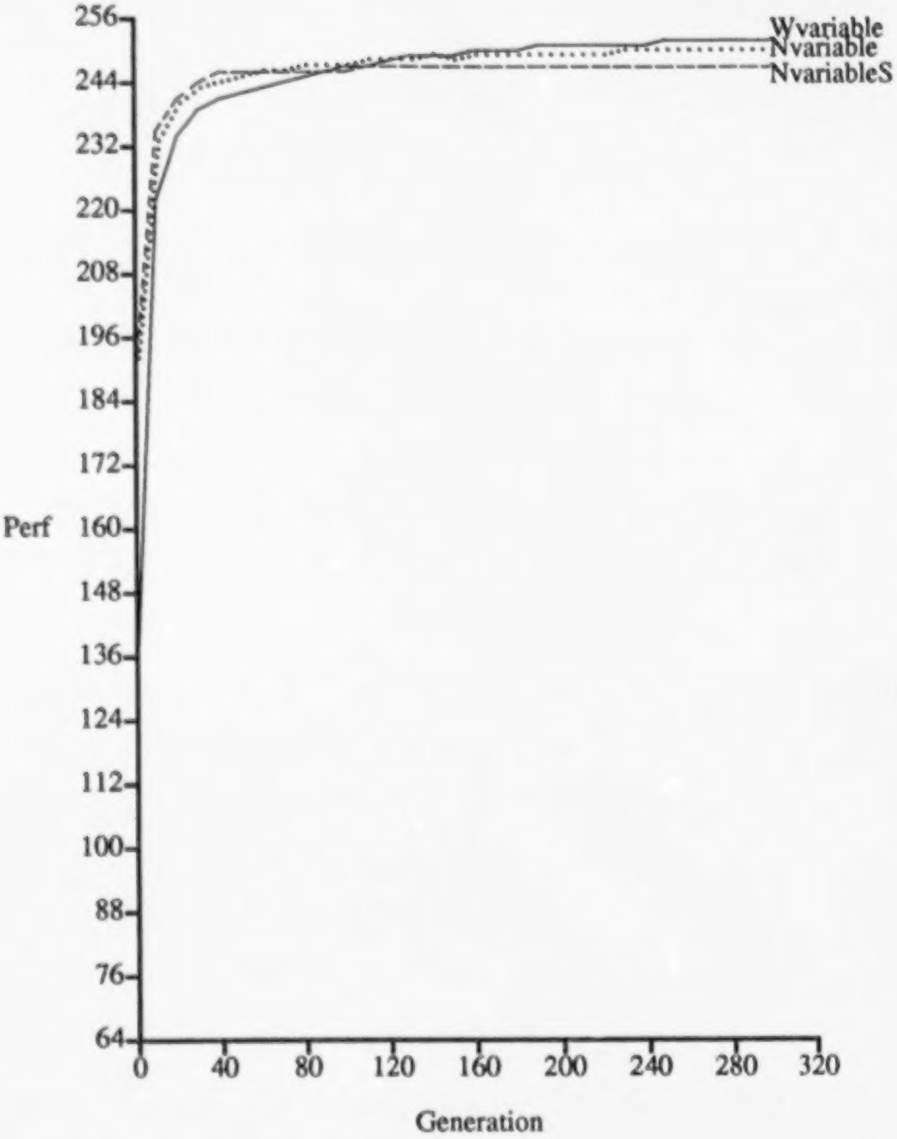

Figure 5.4: f_2 performance, $ps = 64$

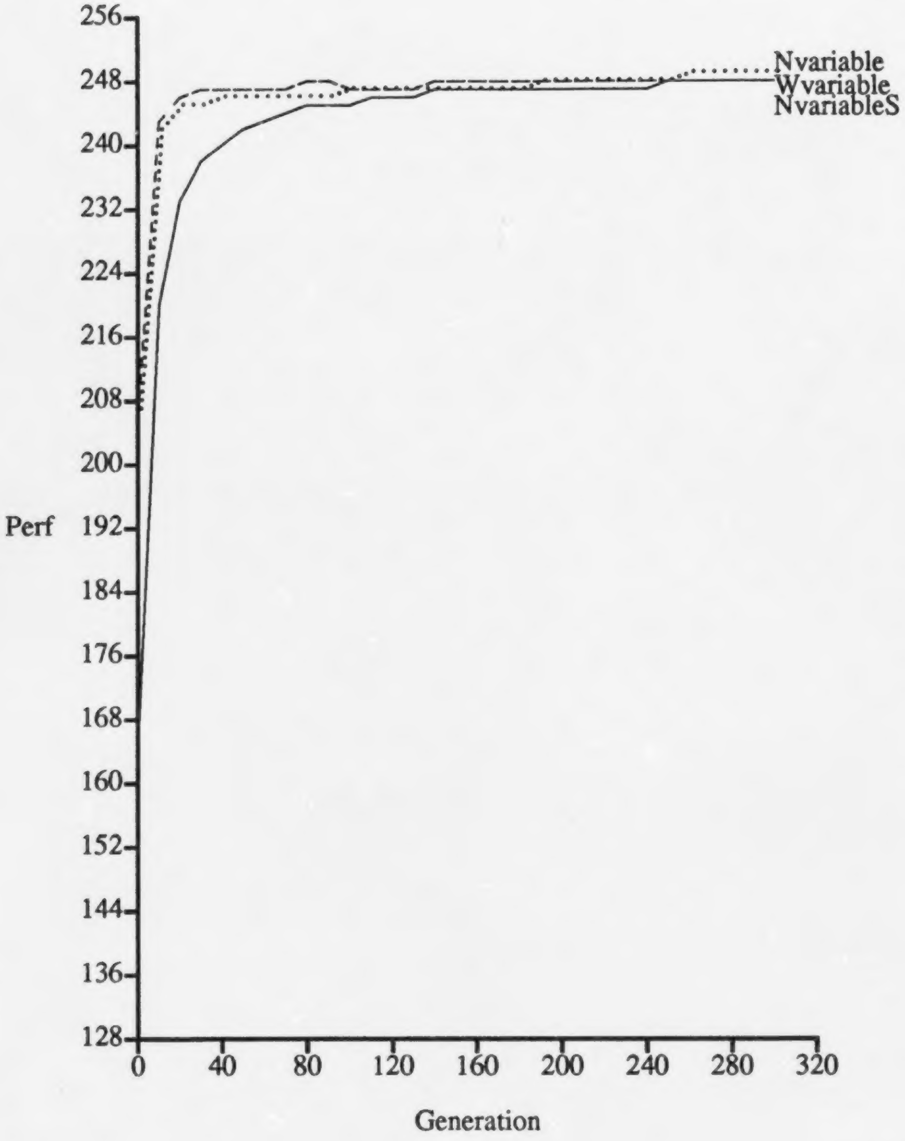

Figure 5.5: f_2 performance, $ps = 128$

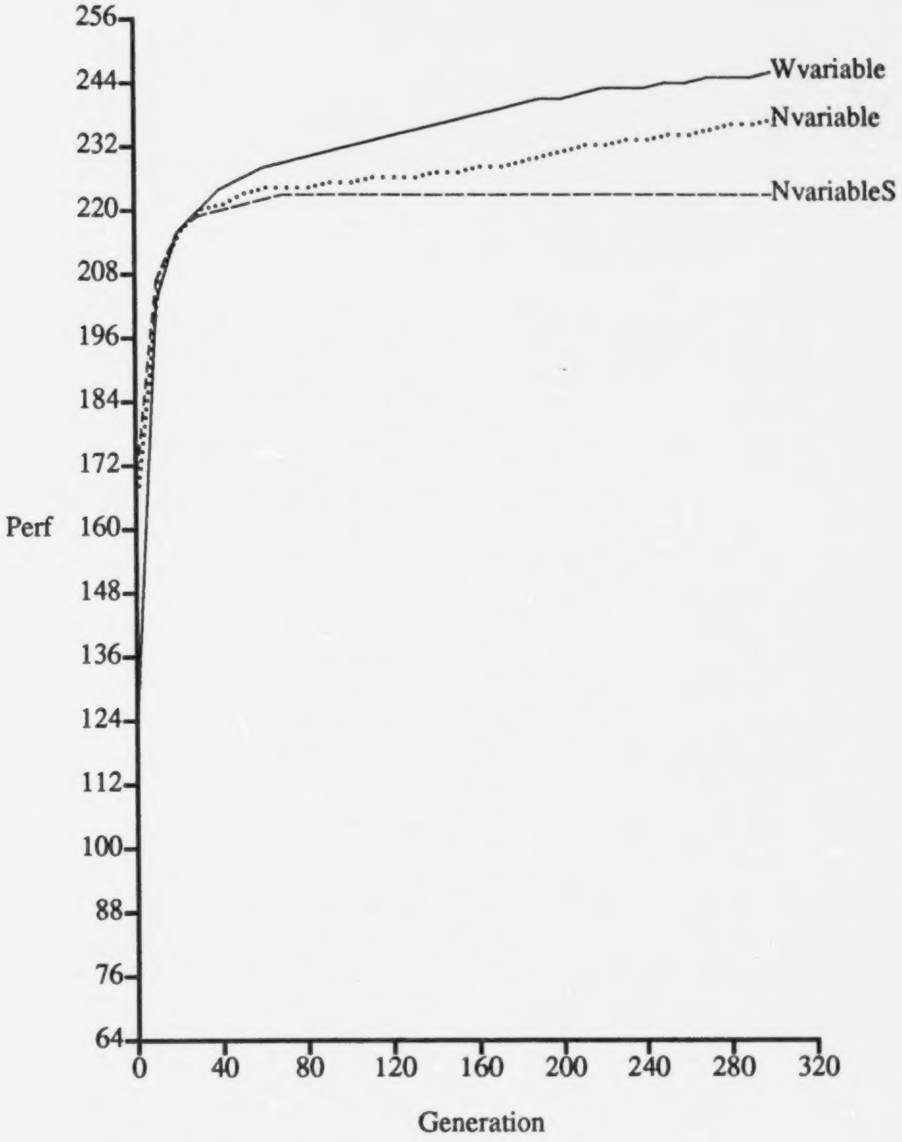

Figure 5.6: f_4 performance, $ps = 64$

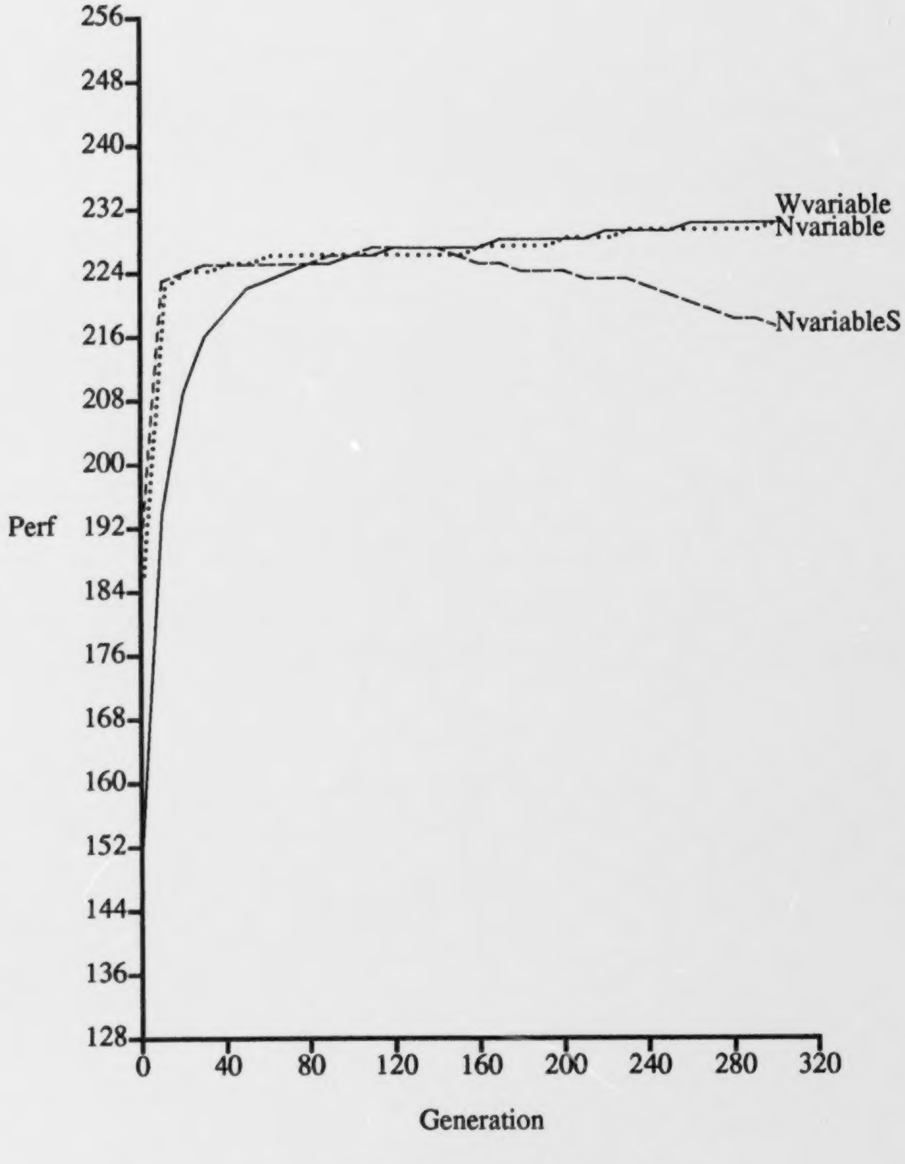

Figure 5.7: f_4 performance, $ps = 128$

7. The experiments for all the above functions show that when a *ps* is too small relative to the problem, the performance of VCS might be worse than those of *Nvariable* and *NvariableS*. Figures 5.8 and 5.9 show two examples. Figure 5.8 illustrates the performances for f_1 with $ps = 10$. Although all the performances shown in the figure are poor, *Wvariable* is worse than *Nvariable*. In Figure 5.9, *Wvariable* is worse than both *Nvariable* and *NvariableS*.

8. Finally, we have noticed that in our experiments, *NvariableS* generally performs poorly. We attribute this problem to the bidding strategy $bid = \frac{k \times u_c \times sp}{2^{l-sp}}$. This bidding strategy caused the problem of "starving the generals" seriously. Classifiers with '#'s have almost no chance to win the bids. Very few subsolution structures with '#'s are built up. Therefore, the disruptive problem is more serious. This point also shows the importance of using good data structures for stable performances.

5.5. Concluding Remarks

Variables are used to change the way in which a set of primitive solutions can be organized. From a data structures view point, variables are usually able to add connections among primitive solutions. It has been shown that when ties among primitive solutions are stronger, the performance is higher and more stable. VCS is a low-level representation framework with some high-level representation capabilities. It narrows the gap between the symbolic learning and genetic classifier system learning in the sense that VCS has much stronger abilities than CS in representing knowledge structures and relationships. Yet VCS still keeps the characteristics of classifier systems in terms of learning mechanisms. Compared with the variable scheme used in Smith's LS-1 [Gol89a, Smi80], the VCS variable scheme is more general. More importantly,

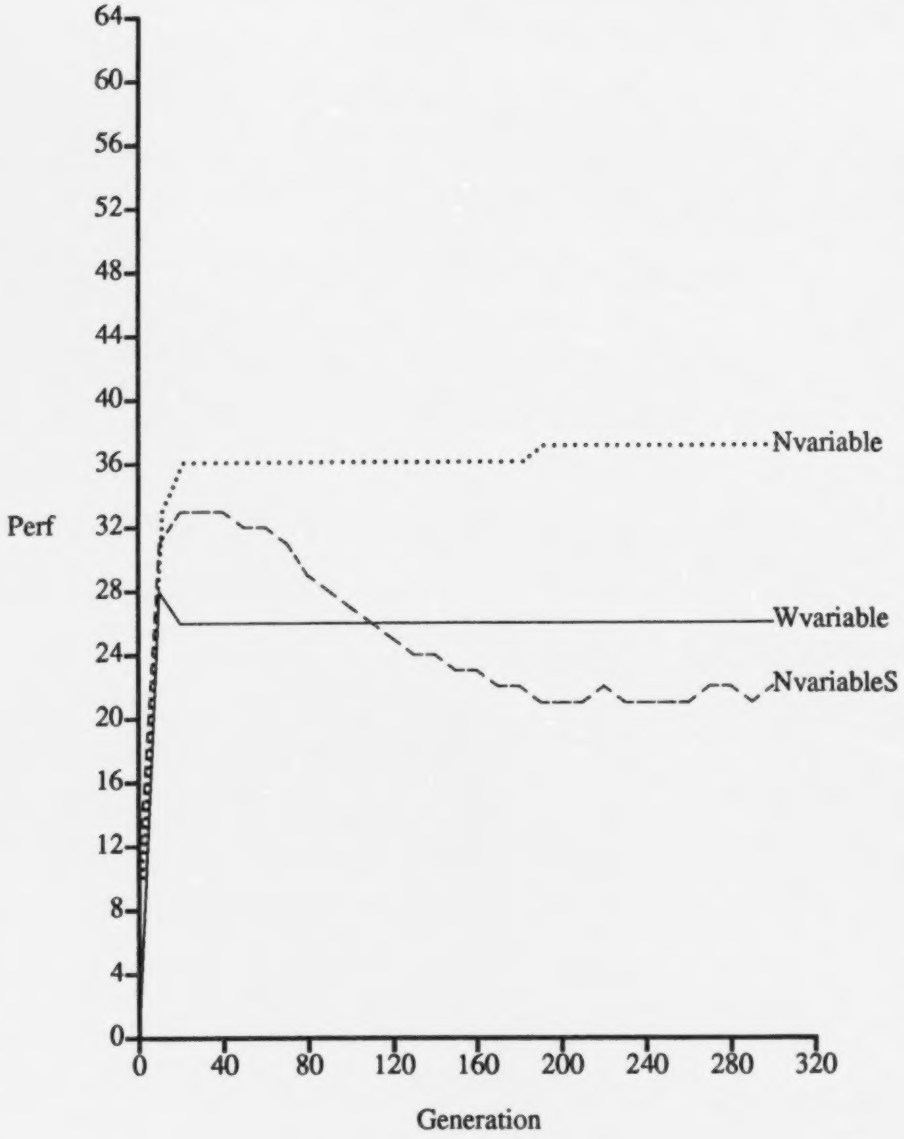

Figure 5.8: f_1 performance, $ps = 10$

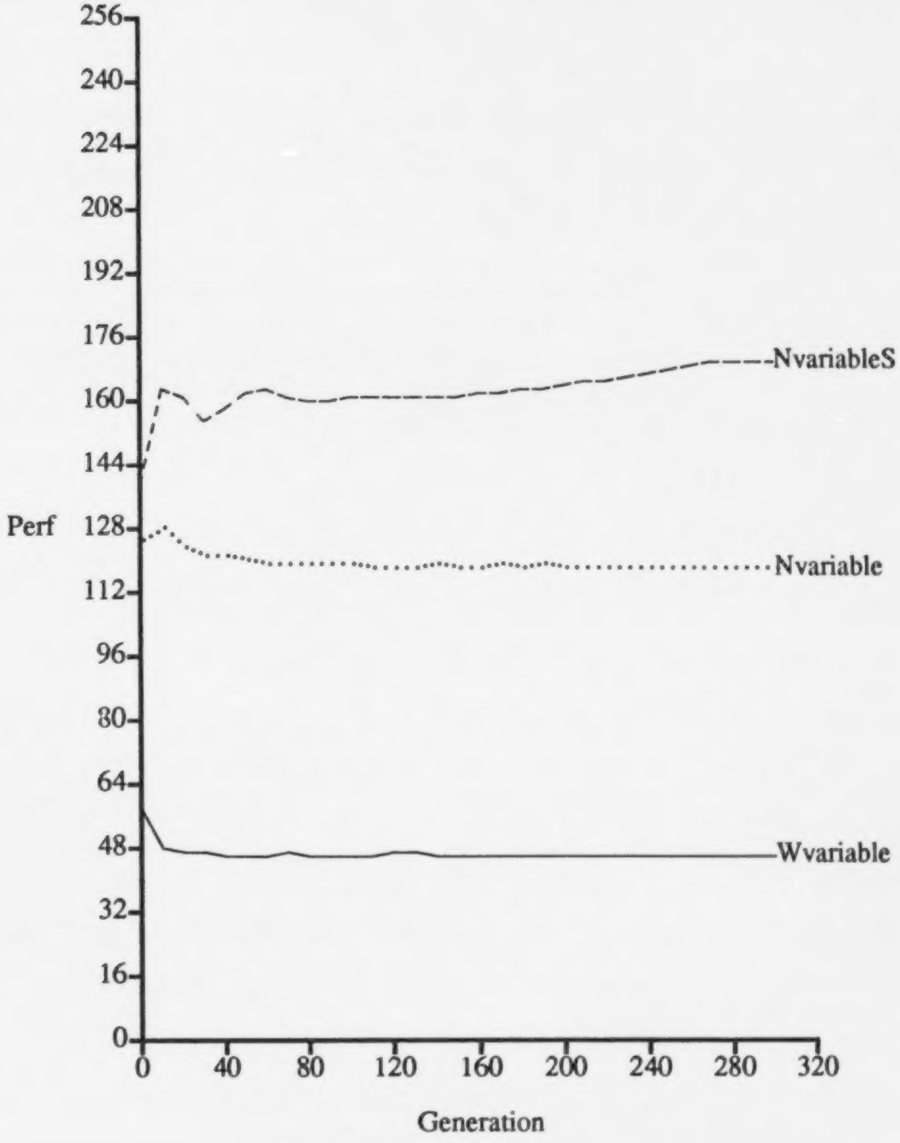

Figure 5.9: f_4 performance, ps = 16

VCS provides a low-level processing framework that is capable of processing with variables. The variable sign "*" in VCS is treated uniformly as other symbols. It is possible to implement learning mechanisms that are used in other frameworks in VCS. Knowledge structures can be built into VCS applications with some model relation rules.

VCS also has promise for use in chunked learning. The traditional view of chunks is that they are the compact representations of several items. A schema is an example of a chunk. The more recent view of chunks is that they could also be data structures representing processes or procedures. From this point of view, functions in conventional programming languages are examples of chunks. The introduction of variables to classifier systems makes rules begin to look like functions. Values are bound to function parameters when a message matches a condition. The action part is just the return value(s) from the function call. Between the condition and action parts is a simple function body. If VCS continues to evolve, making the function describing the mapping of input to output values more sophisticated, then these systems become more like a conventional programming language. The only major difference is the method used for representing values and variables. Maybe this implies that the simple binary representations of CS and VCS should be the next area for evolution. The implementation of chunks as functions would make the knowledge being structured as a lattice rooted in a set of pre-existing primitives [RoN86]. How to detect, encode and decode a chunk in VCS, perhaps using techniques from traditional programming languages, would be an interesting research project.

Theoretically, the properties of VCS are applicable to internal message passing systems. Although there is no internal message passing involved in the experiments in this work, we expect that VCS will help reduce the number of chains needed because of the possible succinct representation brought by variables. However, since variable bindings are done in the scope of a classifier, there is no semantic connections between variables in different classifiers in VCS. Therefore, variables may not be helpful in reducing the length of the chains. Instances represented by a variable support each other's survival in the population. But, more study is needed to answer if variables have the effect of encouraging co-adaptation of the individuals in a chain and how variables affect the formation of chains.

Chapter 6

Conclusions

This chapter contains two sections. Section 6.1 concludes the contributions of this dissertation. Section 6.2 discusses future research directions.

6.1. Contributions of the Dissertation

Over the years, research in machine learning has been pursued with varying degrees of intensity, using different approaches and placing emphasis on different aspects and goals. As a general inductive learning model, genetic classifier learning systems emphasize efficient low-level learning, in which lack of knowledge or incomplete knowledge at the symbolic name level can be compensated by efficient processing of available information. The genetic learning systems have been applied to problems, such as knowledge acquisition and dynamic control, where prior heuristic knowledge needed for solving problems is not available and the environment is noisy.

Although genetic learning exhibits promising properties, its application has been limited by poor performance, consequences of the rule clustering and rule association problems. Different selection strategies and credit apportionment algorithms have been used to solve these problems. However, the results are still not satisfactory.

This dissertation studies the impact of data structures in a framework on the performance and on the resolution of the above problems. The concept of primitive solutions and internal structures of primitive solutions are proposed in this research. Based on these concepts, we suggest that data structures that support the internal structures of sub-solutions help solve the above problems without complicating the systems, such as introducing new parameters. In other words, if a data structure can represent many primitive solutions or sub-solutions in a succinct way, the fates of these primitive solutions or sub-solutions are tightly connected. Therefore co-adaptations are encouraged in such frameworks. Based on the above theory, we tackle the rule clustering and rule association problems by using data structures that would associate primitive solutions or sub-solutions together. Hierarchical and Variable classifier systems are designed and implemented as two general frameworks for genetic-algorithm-based learning. In these frameworks, new data structures are introduced into conventional classifier systems to associate primitive solutions and sub-solutions.

In the hierarchical classifier systems (*HCS*), structural ties are imposed on single classifiers and a new data structure, family, is introduced. Accordingly, the new crossover operators that operate on the families are introduced. In *HCS*, family units make the individual members strongly related by common interests, gains and loss. Therefore, their behaviors tend to be for each other rather than against each other. Hence the probability of sub-solution domination or disruption of the internal structures of sub-solutions are reduced with reasonable computational complexity.

HCS provides a framework that combines the Michigan approach and the Pittsburgh approach, and is a step towards the unification of those two approaches.

The Variable Classifier Systems (*VCS*) uses a different approach than the structural ties method used in *HCS* to introduce better data structures. *VCS* takes advantages of the semantic meaning of variables. Classifiers in *VCS* are better data structures than those in conventional classifier systems. They have stronger representation abilities. Many solutions can be expressed in a much more succinct way, therefore reducing the number of classifiers needed to be found in a search.

VCS provides a model for combining a high-level symbolic approach and a low-level processing approach. Compared with the conventional classifier systems, *VCS* is more capable of describing an abstract world, allowing for the building of models and knowledge structures as in high-level symbolic representation systems. It thus becomes possible to adopt techniques in symbolic approaches into the genetic learning.

Performance wise, *HCS* and *VCS* provide two general genetic learning frameworks which reduce the rule clustering and rule association problems and produce better performance than conventional classifier systems.

From the view point of methodology, the work on *HCS* and *VCS* provides a different perspective of looking into the fundamental problems of classifier systems. The work tries to solve the problems by introducing better data structures. There are two advantages to this approach. First, the introduction of better data structures reduces the dependence of performance on the complementary mechanisms, such as the credit apportionment algorithms, and therefore reduces the scope and complexity of the problems and difficulties involved in the systems. Second, using better data structures does not introduce any new complementary learning mechanisms into the systems.

Another contribution of this dissertation is the study on initial populations. This study provides several general guidelines for designing a better initial population for a problem. Two measures are defined for determining how good a population is and two algorithms for generating better initial populations are proposed.

6.2. Future Research Directions

Hierarchical classifier systems are proposed as an approach to provide better data structures for GABL, and to impose structural ties and enforce dependence among classifiers. In such systems, cooperation and co-adaptation are encouraged. When applying this approach, we suggest that the number of hierarchies needed in a system is problem dependent. For problems with large search and solution spaces, more hierarchies are needed to offer better data structures and organizations. How to choose an optimal number of hierarchies for a problem is an interesting research topic.

The experiment results show that bigger family sizes are needed by larger problems to achieve better performances when only a relatively small population size is available. However, a family size that is too big relative to the problem may have a severe parasite problem and the efficiency would be degraded because of its larger search space. So, rules for choosing a proper family size are important in maximizing the potential of this approach. This dissertation only discussed the family size issue based on the experimental results and offered several general guidelines. More work is needed to provide concrete principles.

Allowing varied family sizes in a population is another interesting issue. The coexistence of different sized families would be amenable to building families according to relations r_i and help in reducing the parasite problem.

Classifiers in the variable classifier systems are better data structures than those in conventional classifier systems. The use of variables increases the symbolic processing ability of genetic learning and makes it possible to combine symbolic and low-level learning into one framework. Building such a framework is a significant task since to solve a real world problem effectively and efficiently, different learning approaches are needed at different stages of a learning [Mic86].

In this dissertation, the hierarchical approach and variable approach are studied and implemented separately. We believe that the combination of these two approaches is a framework that has stronger cooperation among classifiers and stronger representation abilities. More work is needed to verify this.

Theoretically, the properties of HCS and VCS are applicable to internal message passing systems. Although there is no internal message passing in the implementations of HCS and VCS in this work, we believe that HCS has the additional potential of increasing the stability of chains. Particularly, HCS is able to reduce the adverse effect of the payoff delay to the stage-setting classifiers in a chain and thus is especially beneficial to long chains. Experimental work is needed to verify this. More study is needed to determine the impact of variables on the internal structures built through the internal message passing.

Finally, from the methodology view point, further investigation is needed on how better data structures reduce the sensitivity of the performance of genetic learning to the control parameters and other complementary mechanisms such as credit apportionment algorithms.

References

[And86] J. R. Anderson, Knowledge Compilation: the General Learning Mechanism, in *Machine Learning II* , R. S. Michalski, J. G. Carbonell and T. M. Mitchell (ed.), Morgan Kaufmann Publishers, Inc., 1986.

[BeF88] R. K. Belew and S. Forrest, Learning and Programming in Classifier Systems, *Machine Learning 3*, (1988), 193-224, Kluwer Academic Publishers.

[Bet81] A. D. Bethke, *Genetic Algorithms as Function Optimizer*, Ph.D. Thesis, University of Michigan, 1981.

[Boo82] L. B. Booker, *Intelligent Behavior as an Adaptation in the Environment*, Ph.D. Thesis, Technical Report, University of Michigan, 1982.

[Boo89] L. B. Booker, Triggered Rule Discovery in Classifier System, *Third International Conference on Genetic Algorithms*, 1989, 265-274.

[Bri81] A. Brindle, *Genetic Algorithms for Function Optimization*, Ph.D. Thesis, University of Alberta, 1981.

[Car90] J. Carbonell, Introduction: Paradigms for Machine Learning, in *MACHINE LEARNING: Paradigms and Methods*, J. Carbonell (ed.), MIT/Elsevier, 1990, 1-11.

[DeJ75] K. A. DeJong, *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, Ph.D. Thesis, University of Michigan, 1975.

[DeJ87] K. A. DeJong, On Using Genetic Algorithms to Search Program Spaces, *Second International Conference on Genetic Algorithms and Their Applications*, MIT, Cambridge, MA, 1987, 210 - 216.

[DeJ88] K. A. DeJong, Learning With Genetic Algorithms: An Overview, *Machine Learning 3*, (1988), 121-138, Kluwer Academic Publishers.

[For82] C. L. Forgy, Rete: A Fast Algorithm For the Many Pattern/Many Object Pattern Match Problem, *Artificial Intelligence 19*, (1982), 17 - 37.

[For85] S. Forrest, *A Study of Parallelism in the Classifier System and its Application to Classification in KL-ONE Semantic Networks*, Ph.D. Thesis, University of Michigan, 1985.

[Gil85] A. M. Gillies, Machine Learning Procedures for Generating Image Domain Features Detectors, *Doctoral Dissertation*, Ann Arbor, 1985.

[Gol83] D. E. Goldberg, *Computer-Aided Gas Pipeline Operation Using Genetic Algorithms and Rule Learning*, Ph.D. Thesis, University of Michigan, 1983.

[Gol89a] D. E. Goldberg, *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison-Wesley Publishing Company, 1989.

[Gol89b] D. E. Goldberg, Sizing Populations for Serial and Parallel Genetic Algorithms, *Proceedings of the Third International Conference on Genetic Algorithms*, 1989, 70-80.

[Gre87] J. J. Grefenstette, Incorporating Problem Specific Knowledge into Genetic Algorithms, *Genetic Algorithms and Simulated Annealing*, 1987, 42 - 60.

[Hol75] J. H. Holland, *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, 1975.

[HoR78] J. H. Holland and J. S. Reitman, Cognitive Systems Based on Adaptive Algorithms, in *Pattern Directed Inference Systems*, D. A. Waterman and F. Hayes-Roth (ed.), Academic Press, New York, 1978, 313-329.

[Hol86] J. H. Holland, Escaping Brittleness: The Possibilities of General Purpose Learning Algorithms Applied to Parallel Rule-Based Systems, in *Machine Learning II*, R. S. Michalski, J. G. Carbonell and T. M. Mitchell (ed.), 1986.

[HHN86] J. H. Holland, K. J. Holyoak, R. E. Nisbett and P. R. Thagard, *Induction: Processes of Inference, Learning, and Discovery*, MIT Press, Cambridge, 1986.

[HoU69] J. E. Hopcroft and J. D. Ullman, *Formal Language and Their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

[Jac86] Peter Jackson, *Introduction to Expert Systems*, Addison-Wesley Publishing Company, 1986.

[LRN86] John E. Laird, Paul S. Rosenbloom and Allen Newell, Chunking in SOAR: The Anatomy of a General Learning Mechanism, *Machine Learning 1*, (1986), 11 - 46, Kluwer Academic Publishers.

[Mic86] Ryszard S. Michalski, Understanding the Nature of Learning: Issues and Research Directions, in *Machine Learning II*, R. S. Michalski, J. G. Carbonell and T. M. Mitchell (ed.), Morgan Kaufmann Publishers, Inc., 1986.

[Nil71] Nils J. Nilsson, *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill Book Company, 1971.

[Nil83] Nils J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Co., 1983.

[Nor86] Donald A. Norman, Reflections on Cognition and Parallel Distributed Processing, *Parallel Distributed Processing II*, 1986.

[Qui83] J. R. Quinlan, Learning Efficient Classification Procedures and their

Application to Chess End Games, in *Machine Learning I* , R. S. Michalski, J. G. Carbonell and T. M. Mitchell (ed.), 1983.

[Qui90] J. R. Quinlan, Probabilistic Decision Trees, in *Machine Learning III*, Yves Kodratoff and Ryszard Michalski (ed.), Morgan Kaufmann Publishers, Inc., 1990, 140-153.

[Ric83] Elaine Rich, *Artificial Intelligence*, McGraw - Hill, Inc., 1983.

[Rio86] R. L. Riolo, LETSEQ: An Implementation of the CFS-C Classifier System in a Task Domain that Predicts Letter Sequences, *Technical Report, University of Michigan*, 1986.

[Rio89] R. L. Riolo, The Emergence of Default Hierarchies in Learning Classifier Systems, *Proceedings of the Third International Conference on Genetic Algorithms*, 1989, 322 - 327.

[RoN86] Paul S. Rosenbloom and Allen Newell, The Chunking of Goal Hierarchies: A Generalized Model of Practice, in *Machine Learning II* , R. S. Michalski, J. G. Carbonell and T. M. Mitchell (ed.), Morgan Kaufmann Publishers, Inc., 1986.

[RLN87] Paul S. Rosenbloom, John E. Laird, Allen Newell and Robert McCarl, A Preliminary Analysis of the Soar Architecture as a Basis for General Intelligence, *Draft: Proceedings MIT Workshop on Foundations of Artificial Intelligence*, 1987.

[RHM86] D. E. Rumelhart, G. E. Hinton and J. L. Mcclelland, A General Framework for Parallel Distributed Processing, *Parallel Distributed Processing I*, 1986.

[ScG85] J. D. Schaffer and J. J. Grefenstette, Multi-objective Learning via Genetic Algorithms, *9th International Joint Conference on Artificial Intelligence*, 1985, 593-595.

[SCE89] J. D. Schaffer, R. A. Caruana, L. J. Eshelman and R. Das, A Study of Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization, *Proceedings of the Third International Conference on Genetic Algorithms*, 1989, 51 - 60.

[Sch88] D. Schuurmans, Representation and Selection Techniques for Genetic Learning Systems, M.Sc. Thesis, University of Alberta, 1988.

[ShS89] L. Shu and J. Schaeffer, VCS: Variable Classifier Systems, *Proceedings of The Third International Conference on Genetic Algorithms*, 1989.

[ShS91] L. Shu and J. Schaeffer, Adding Hierarchies to Classifier Systems, *Proceedings of The Forth International Conference on Genetic Algorithms*, 1991.

[Smi80] S. F. Smith, *A Learning System Based on Genetic Adaptive Algorithms*, Ph.D. Thesis, University of Pittsburgh, 1980.

[SmG90] R. E. Smith and D. E. Goldberg, Reinforcement Learning With Classifier Systems, *AI, Simulation and Planning in High Autonomy Systems*, Los Alamitos, 1990, 184-192.

[Tsy73] Y. Z. Tsypkin, *Foundations of the Theory of Learning Systems*, Academic Press, New York and London, 1973.

[Wil86] S. W. Wilson, Knowledge Growth in an Artificial Animal, in *Adaptive and Learning Systems: Theory and Applications*, K. S. Narendra (ed.), Plenum, New York, 1986, 255-264.

[Wil88] S. W. Wilson, Bid Competition and Specificity Reconsidered., *Complex Systems 2(6)*, (1988), 705-723.

[WiG89] S. W. Wilson and D. E. Goldberg, A Critical Review of Classifier Systems, *Proceedings of the Third International Conference on Genetic Algorithms*, 1989, 244-254.

[ZhG89] H. Zhou and J. J. Grefenstette, Learning by Analogy in Genetic Classifier Systems, *Proceedings of the Third International Conference on Genetic Algorithms*, 1989, 291-298.

Appendix 1

The following is the definition for the minimal Boolean function of eight variables used in this work.

$f(x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7) =$

(NOT(x_0) AND x_1 AND NOT(x_2) AND NOT(x_4) AND x_5 AND NOT(x_6) AND x_7) OR

(NOT(x_0) AND NOT(x_1) AND NOT(x_2) AND NOT(x_3) AND x_4 AND x_5 OR

(NOT(x_0) AND x_1 AND NOT(x_2) AND NOT(x_3) AND x_4 AND x_6 AND NOT(x_7)) OR

(NOT(x_0) AND NOT(x_1) AND NOT(x_2) AND NOT(x_3) ANDx_4 AND NOT(x_6) AND x_7) OR

(NOT(x_0) AND NOT(x_1) AND x_2 AND NOT(x_3) AND NOT(x_4) AND x_6 AND x_7) OR

NOT(x_1) AND x_2 AND x_3 AND NOT(x_5) AND NOT(x_6) AND x_7 AND) OR

(NOT(x_0) AND x_1 AND NOT(x_3) AND x_4 AND NOT(x_5) AND x_6) OR

(x_0 AND x_2 AND x_3 AND x_4 AND NOT(x_5)) OR

(NOT(x_0) AND NOT(x_1) AND x_2 AND NOT(x_4) AND NOT(x_5) AND x_6) OR

(x_0 AND NOT(x_2) AND x_3 AND x_4 AND x_5 AND NOT(x_6) AND NOT(x_7)) OR

(x_0 AND NOT(x_2) AND x_3 AND NOT(x_4) AND x_5 AND x_6 AND NOT(x_7)) OR

(x_0 AND NOT(x_1) AND x_2 AND x_3 AND NOT(x_4) AND NOT(x_6) AND x_7) OR

(x_1 AND x_2 AND NOT(x_3) AND x_4 AND x_6 AND x_7) OR

(x_0 AND NOT(x_2) AND x_3 AND NOT(x_4) AND NOT(x_5) AND NOT(x_6) AND NOT(x_7)) OR

(x_0 AND x_2 AND NOT(x_3) AND x_4 AND x_5 AND x_6 AND NOT(x_7)) OR

(x_0 AND NOT(x_1) AND NOT(x_2) AND NOT(x_4) AND x_6 AND x_7) OR

(NOT(x_0) AND x_1 AND NOT(x_3) AND NOT(x_5) AND x_6 AND x_7) OR

(NOT(x_0) AND NOT(x_1) AND x_2 AND NOT(x_3) AND x_4 AND x_6 AND NOT(x_7)) OR

(x_0 AND NOT(x_1) AND x_2 AND x_3 AND x_4 AND x_6 AND NOT(x_7)) OR

(x_0 AND x_1 AND x_3 AND NOT(x_5) AND NOT(x_6) AND x_7 AND) OR

(x_1 AND NOT(x_2) AND x_3 AND NOT(x_4) AND NOT(x_5) AND x_7) OR

(NOT(x_1) AND NOT(x_2) AND x_3 AND x_4 AND NOT(x_5) AND NOT(x_6)) OR

(NOT(x_0) AND NOT(x_1) AND NOT(x_3) AND NOT(x_4) AND NOT(x_5) AND x_7) OR

(x_1 AND x_3 AND NOT(x_4) AND NOT(x_5) AND x_6 AND NOT(x_7)) OR

(x_0 AND NOT(x_1) AND x_2 AND x_4 AND NOT(x_5) AND x_7) OR

(x_0 AND x_2 AND NOT(x_4) AND NOT(x_5) AND x_6 AND NOT(x_7)) OR

(NOT(x_0) AND x_1 AND x_2 AND NOT(x_3) AND x_4 AND x_5 AND NOT(x_6) AND NOT(x_7)) OR

(x_0 AND x_1 AND x_2 AND x_3 AND x_5 AND NOT(x_6) AND NOT(x_7)) OR
(NOT(x_0) AND NOT(x_1) AND x_2 AND NOT(x_3) AND
NOT(x_4) AND x_5 AND NOT(x_6) AND NOT(x_7)) OR
(NOT(x_0) AND NOT(x_1) AND x_3 AND x_4 AND x_5 AND NOT(x_6) AND NOT(x_7)) OR
(x_0 AND NOT(x_2) AND NOT(x_3) AND NOT(x_4) AND x_5 AND NOT(x_6)) OR
(x_0 AND NOT(x_3) AND NOT(x_4) AND x_5 AND NOT(x_6) AND x_7) OR
(x_0 AND x_1 AND x_2 AND x_3 AND x_4 AND NOT(x_6)) OR
(NOT(x_0) AND NOT(x_1) AND NOT(x_3) AND x_4 AND x_5 AND x_7) OR
(NOT(x_0) AND x_1 AND NOT(x_2) AND x_3 AND NOT(x_4) AND x_5 AND NOT(x_6)) OR
(x_0 AND NOT(x_2) AND NOT(x_3) AND x_5 AND NOT(x_6) AND x_7) OR
(NOT(x_0) AND x_1 AND x_2 AND NOT(x_3) AND NOT(x_4) AND x_5 AND x_6 AND NOT(x_7)) OR
(NOT(x_0) AND x_1 AND x_2 AND x_3 AND x_4 AND x_5 AND x_6 AND NOT(x_7)) OR
(NOT(x_0) AND x_1 AND x_3 AND NOT(x_4) AND x_5 AND NOT(x_6) AND x_7) OR
(NOT(x_0) AND x_1 AND NOT(x_2) AND x_3 AND x_5 AND NOT(x_6) AND x_7) OR
(x_0 AND x_2 AND x_4 AND NOT(x_5) AND NOT(x_6) AND NOT(x_7)) OR
(NOT(x_0) AND NOT(x_1) AND x_3 AND NOT(x_4) AND x_6 AND NOT(x_7)) OR
(NOT(x_0) AND x_1 AND x_2 AND NOT(x_3) AND
NOT(x_4) AND NOT(x_5) AND NOT(x_6) AND NOT(x_7)) OR
(x_1 AND x_2 AND x_3 AND x_4 AND NOT(x_5) AND NOT(x_6) AND NOT(x_7)) OR
(NOT(x_0) AND NOT(x_1) AND NOT(x_2) AND x_5 AND x_6 AND NOT(x_7)) OR
(NOT(x_0) AND NOT(x_2) AND x_3 AND NOT(x_4) AND x_5 AND x_7) OR
(x_0 AND x_1 AND x_2 AND x_3 AND NOT(x_4) AND x_6 AND x_7) OR
(NOT(x_0) AND x_1 AND NOT(x_2) AND NOT(x_3) AND x_4 AND NOT(x_5)) OR
(NOT(x_0) AND NOT(x_1) AND x_2 AND x_4 AND x_5 AND x_6 AND x_7) OR
(x_0 AND NOT(x_2) AND NOT(x_3) AND NOT(x_4) AND x_7) OR
(x_0 AND NOT(x_2) AND x_3 AND x_4 AND x_5 AND x_6 AND x_7) OR
(NOT(x_0) AND x_1 AND NOT(x_3) AND x_4 AND NOT(x_5) AND x_7) OR
(NOT(x_0) AND NOT(x_1) AND x_2 AND x_3 AND NOT(x_4) AND NOT(x_5) AND) OR
(NOT(x_0) AND NOT(x_1) AND NOT(x_2) AND NOT(x_3) AND
NOT(x_4) AND NOT(x_5) AND NOT(x_6)) OR
(x_0 AND NOT(x_2) AND NOT(x_3) AND NOT(x_5) AND x_6 AND NOT(x_7)) OR
(NOT(x_0) AND x_1 AND NOT(x_2) AND NOT(x_5) AND x_6 AND NOT(x_7)) OR
(x_1 AND x_2 AND x_4 AND NOT(x_5) AND x_6 AND x_7) OR
(NOT(x_0) AND NOT(x_1) AND NOT(x_2) AND x_3 AND x_4 AND NOT(x_5) AND x_7).

Appendix 2

The following is an initial population for learning function f_1 in a classifier system.

```
##0#/1
###1/1
10##/0
#11#/0
01#1/1
0#11/1
1##0/0
01#1/1
1###/1
0111/1
01##/0
#0#0/0
0001/1
#001/0
#10#/0
###0/0
111#/0
0###/0
0#10/0
10#0/1
0##0/1
1100/0
001#/0
1###/0
##10/1
1#00/1
1#11/1
0111/0
#011/0
#000/1
11##/1
1111/1
#0##/0
1100/0
1###/0
```

100#/1
##1#/0
100#/0
#1#1/0
0#0#/1
11##/1
#001/0
1###/1
#0##/0
##00/0
0#01/1
010#/1
0#0#/0
#00#/1
##11/0
#100/0
00#1/0
0011/1
#01#/0
##01/1
011#/0
1#11/0
#111/0
1010/0
011#/1

Appendix 3

The following is an initial population for learning function f_1 in an *HSC* with a family size of 2.

{##0#/1, ###1/1}
{10##/0, #11#/0}
{01#1/1, 0#11/1}
{1##0/0, 01#1/1}
{1###/1, 0111/1}
{01##/0, #0#0/0}
{0001/1, #001/0}
{#10#/0, ###0/0}
{111#/0, 0###/0}
{0#10/0, 10#0/1}
{0##0/1, 1100/0}
{001#/0, 1###/0}
{##10/1, 1#00/1}
{1#11/1, 0111/0}
{#011/0, #000/1}
{11##/1, 1111/1}
{#0##/0, 1100/0}
{1###/0, 100#/1}
{##1#/0, 100#/0}
{#1#1/0, 0#0#/1}
{11##/1, #001/0}
{1###/1, #0##/0}
{##00/0, 0#01/1}
{010#/1, 0#0#/0}
{#00#/1, ##11/0}
{#100/0, 00#1/0}
{0011/1, #01#/0}
{##01/1, 011#/0}
{1#11/0, #111/0}
{1010/0, 011#/1}